An Improved Method for Database Design

陳智威

CHAN, Chi Wai Alan

A dissertation submitted in partial
fulfillment of the requirements for the degree of
Master of Philosophy
in
Systems Engineering and Engineering Management

April 2004

Contents

Abstract

The problem with Object-Oriented databases is that they are rich in functionality but poor in performance. This limitation is more apparent especially when processing user's transaction requests for update and retrieval of information. This process incurs high database operating cost and reduces the performance of OODBs. A better organization of the physical storage structure must be found to improve performance.

The literature has shown that the physical storage design problem for OODB is a combinational optimization problem. This physical organization problem is suggested to solve with GA, but the traditional GA performance is poor in terms of converging rate and it takes a long computational time to reach a solution.

Traditional GA is dissatisfactory with low converging rate and long computational time. In our research, we tested various GA operators' performances in solving this design problem. The characteristics of GA operators in the literature are studied and adopted to solve the database design problem. For instance, Enhanced Crossover Operator and Propagation Adjustment method are proposed for better GA performance.

We run the experiments on a PC running on Window 2000 Operation Systems. Experiment results reveal that these operators and methods finding the converging rate and reduce computational time in a solution. Solving this physical database design for high-complexity OODBs application with GA at higher converging rate can meet nowadays requirements of frequent changes of users' transaction pattern.

論文概述

目標導向數據庫(Object-Oriented Database)一向被認爲是功能多但表現參差。當使用者查詢及更新數據時該缺點會更明顯。因處理大量的數據令運作成本增加，降低了目標導向數據庫的表現。 因此有必要選出一個恰當的有形儲存架構，改善數據庫該方面的表現。

在文獻中曾經提及有形儲存結構問題是定義爲組合最優化問題，這有形儲存結構問題擬用遺傳算法(Genetic Algorithm，簡稱 GA)來解決，但傳統的 GA 算法的表現都有不足之處，在於其計算步向最優方案較慢和運算時間較長。

由於傳統的 GA 算法會聚率慢和運算時間長，經考慮問題的特性後，本論文嘗試測試各種 GA 算子的表現，務求解決該項儲存結構問題，經過我們研究文獻所提及之 GA 算法，並採納合適的應用在實驗上。 例如，我們提出了改良交叉算子(Enhanced Crossover Operator)和延伸調整方法(Propagation Adjustment Methods)去改善 GA 表現。實驗結果顯示以上的改良方法能使解決方案加快步向最優化和減少運算時間。

Acknowledgements

During the writing of this thesis, the author received many helps from the persons whom I would like to thank.

First of all, I would like to sincerely thank my supervisor, Professor C. H. Cheng, for his guidance and support in my research. He gave me many useful suggestions and direction. This thesis would not be completed without his help.

Secondly, I wish to thank my internal reviewers, Professor Wai Lam and Professor Christopher Yang, for their careful reading of my thesis and their comments.

I would also like to thank the staff of the Department of Systems Engineering and Engineering Management for their help. A special thanks to my friends, Dr. Chun Hung Chui, Winnie Yip, Matthew Tam, Johnny Cheng and Jacky Wong, whom they gave me encouragement, enduring support during my courses and thesis writing.

Finally, I wish to thank my family for all their love, understanding and caring. I would also give my thanks to them.

List of Figures

List of Tables

Chapter 1

1. Introduction

1.1. Object-oriented databases

Object-oriented databases (OODBs) are the state-of-the-art database technology in the 1990s, just as relational databases for the 1980s. Current relational databases are appropriate for business applications, but inappropriate for complex, large-scale engineering tasks because they are weak in complex object modeling and programming. These applications often require high database performance. Hence, advanced systems are needed to provide enhanced database management capabilities to overcome the limitations of relational and other record-oriented data models such as hierarchical and network. Complex, large-scale database applications such as Computer-Aid Design (CAD) [34] and hypermedia [18] have emerged. However, the performance of data access is still a

major concern [4, 29, 30].

OODBs are commonly believed to be 'rich in functionality but poor in performance', especially when many classes are involved [36]. A major consideration in determining the performance of OODBs is the physical storage structure [25]. There has been little research on this issue.

The implementers of object-oriented database management systems (OODBMS) use various models of the physical structures [25]. These physical implementation models differ in the manner in which individual objects and instance variables are stored in a database. Individual objects stored in a class incur higher operating cost if they must be accessed from the secondary memory. The number of data accesses to classes needed to satisfy users' requests is a pseudo measure for database operating cost. With the aim to improve OODB by reducing such database operational cost, the design of the physical storage structure intends to organize data in such a way that reducing frequent transactions access to classes for data being stored in the secondary memory. OODB design in terms of physical organization of these instances in the database can result in the least database operating cost. The physical storage design for OODB is a combinational optimization problem and has been solved with Genetic Algorithm (GA) [32]. However, Gorla's GA suffers a low convergence rate and long computation time. In this research, we propose enhanced GA

operators and constraint handling methods, the Enhanced Crossover Operator (ECO) and Enhanced Adjustment Methods (EAMs) to solve physical storage design problem with GA. Experiments are conduced to show how these operators and methods address the limitations of Gorla's GA.

1.2. Object-oriented Data Model

The object-oriented data model is more complex than the relational model, because there is no universally accepted data model for OODB [24]. The object-orientation is not based on a formal mathematical model like the relational model [5]. Object-oriented data models support the notions of classes, subclasses, class hierarchies, and objects [34]. In OODB, each entity is an object. Objects include instance variables (similar to attributes in the relational model that describe the state of the object. Objects also include methods, which contain instructions to manipulate the object or return object state. Object with the same set of instance variables and methods are grouped into a class, and the objects are called instances of the class. Similar classes are grouped into superclasses, thus forming superclass-subclass relationship.

1.3. Class and Object Instances

A class is the language constructs most commonly used to define abstract data types in object-oriented programming languages. A class incorporates the definition of the structure as well as the operations of the abstract data type. Elements that belong to the collection of objects described in a class are called instances of the class. A class definition includes at least the *class name*, the *class method* for manipulating the instances of the class, and the internal representation. A class definition includes the code that implements the class's interface operators plus descriptions of the internal representation of objects (object states) in that class. Among these terms, internal representation, which captures the values of various states of the class instances, seems to be confusing in its definition. An example of internal representation is internal instance variables, which are the set of documents contained in a folder such as the salary of a staff.

The values of the variables in the internal representation of the instances of the class pertain to individual objects. For instances, John's internal representation consists of his description (name, address, etc.) and the accounts and active orders he has handled. The aggregation of the full set of these values captures the state of John as an instance of SalesPerson.

1.4. Inheritance

Object-oriented database systems provide the modeling of real-world applications as closely as possible. Apart from its achievement in software reusability and software extensibility, inheritance is important in OODB functionality. It enables the construction of new object types and software modules (e.g. classes) on top of an existing hierarchy of modules. New classes can inherit both the behavior such as methods and the representation such as instance variables from existing classes. Inheriting behavior enables code sharing among software modules. Inheriting representation enables structure sharing among data objects. The combination of these two types of inheritance provides a powerful modeling and software development strategy. Inheritance also provides a natural mechanism for organizing information. It "taxonomies" objects into well-defined inheritance hierarchies.

Inheritance introduces some capability for OODB to handle the data model with high-complexity, such as CAD [34] and expert systems [31]. Indeed, some facets of inheritance are summarized as below [43]:

1. Inheritance allows programmers to construct more specialized systems from existing class hierarchies, so that existing software can refer to

and reuse existing modules.

2. In the inheritance, subclass-visible is the third alternative other than public and private instance variables that allow the direct manipulation of instance variables in the superclass. This is important because instance variable stored in superclass can be accessed by its subclass RETRIEVAL or UPDATE transaction.
3. The class of an object describes its structure by specifying the object's instance variables. Classes can be specialized by extending their representation or behavior. Classes can also be specialized by restricting the representation or operations of existing classes. Instances of a subclass must retain the same type of information as instances of their superclass. One way to achieve this is to inherit the instance variables of the superclass directly and allow methods in the subclass to access and manipulate the instance variables or its superclasses without any constraints. In this scheme, each subclass declares the additional instance variables that it introduces as specialization or extension.
4. Class inheritance supports the ability of one class to inherit representation and method from another class. Objects in an inherited class delegate messages to one another, thereby "inheriting" methods or values stored in other objects.
5. Multiple inheritance means a class inherits from more than one

immediate parent.

1.5. Constraint

Constraint tests the correctness or completeness of the abstract data type, which represents a "type" of objects. This "type" of object being tested in database physical design is the inheritance of object-oriented database structure.

Access and update constraint routines are executed when manipulating instances of the abstract data type. These constraint routines are incorporated into the definition of the class. They may be associated with either the object instance as a whole or particular instance variable of the object. This is similar to the notion of integrity constraints in databases. The integrity constraint might specify, for example, that an office worker's salary should not exceed that of his or her manager. Every time the salary of an office worker is updated, the system checks the constraint. When this constraint is violated, an error results, and the system will reject the update. This is similar to the checking of inheritance constraint that is to be discussed in Chapter 3. With object-oriented databases the system can support some integrity constraints for object states directly through, for instance, inheritance constraint.

1.6. Physical Design for OODB Storage

A methodology was developed to determine OODB storage structure. With the aim to meet users' database requirements in the most economical manner, the physical design for OODB storage is to determine either that an instance variable should be inherited and stored in a subclass or the otherwise in a superclass. Instance variables can be inherited from both their direct superclass and indirect superclass. The latter is an instance variable can be inherited by a subclass from its superclass(es). Gorla [32] studied that the possible number of database designs can be very large even with a moderate number of classes and home variables. For example, there are C classes with an average of A home variables per class and a fraction f of the classes are the root-classes, then the possible number of storage structures are at least $2^{AC(1-f)}$ possible physical arrangements of the instance variables. The total home instance variables are A * C and the number of home variables out of them are A * C (1 – f). Even for a small OODB logical schema with five home instance variables per class, and leaf classes constituting 20 percent of the total number of classes, there are at least 2^{20}, or 32 million possible storage structures. It is very difficult to determine the best solution, or optimum, in a reasonable amount of time even for a small problem.

1.7. Problem Description

The object-orientation is more complex than the relational model, but there is no universally accepted data model for OODB [32]. In the database literature, there are several relevant relational database design techniques could be adopted for OODBs to enhance database performance. The file design and accessing techniques for OODBs are similar to those for relational databases. Data fragmentation is used in relational databases for performance improvement. Physical storage structure is one of the major considerations in determination of performance in OODBs, because minimal access to instances is assumed to be synonymous to the reduction of response time.

In this research, we adapted the data fragmentation in relational databases to OODBs for performance improvement. We attempted to determine the lowest operating cost of physical storage structure, or data fragmentation, based on user's information need so that the transaction processing time is reduced.

The problem can be stated as follows: given a logical OODB schema and a set of user retrieval and update requests, determine the

storage structure that yields the minimum database operating cost. The objective is to determine which instance variables should be inherited (and stored) in which class of the OODB schema. The evaluation criterion is the total number of instances accessed to process all the users' retrieval and update requirements, which is a pseudo measure for database operating cost. Since I/O access cost is the major cost in data intensive commercial applications, similar evaluation functions were used in previous research in relational database [37, 41]. In this research, we assume that the instances are stored in all the applicable classes and subclasses. Thus the sub problem can be stated as follows: given a logical OODB schema and user retrieval and update requests, determine an efficient storage structure in terms of which instance variables should be inherited (and stored) in each class; the proposed design results in fewest data accesses.

Physical organization problem for OODB has been solved using GA by N.Gorla [32]. Gorla used Edge-to-edge crossover operator and two adjustment methods, forward and backward adjustments, preserves edge relation information during the generations but limited the searching set of feasible solutions to the combination of edges in solution pool. Thus, Gorla's approach using GA is suffering from low converging rate and long computational time when handling large number of classes. With the aim to exceed this limitation, an effective cross breeding operator and handling

method for infeasible solutions are important to physical storage design using GA. Especially, when users' transaction requests are frequently changed. Enhanced crossover operator should accelerate the converging rate and reduce computational time in solving for feasible solutions. Effective GA operators can help to compute for the optimal physical storage design in shorter time.

1.8. Genetic Algorithm

Our research makes use of Genetic Algorithm. GA was proposed by John Holland [22] in 1975. GA has since become a topic of active research [9] and has been successfully applied in solving some well-known complicated problems such as optimization of gas pipeline [8], Blind Knapsack problem [10], etc. GA is an adaptive method which is based on genetic processes of biological organisms. Over many generations, natural populations evolve according to the principles of natural selection and "survival of the fittest". By mimicking this process, GA is able to "evolve" solutions to real world problems, if it has been suitably encoded. For example, GA can be used to design bridge structures for maximum strength/weight ratio, or to determine the least wasteful layout for cutting shapes from a piece of cloth. It can also be used for online process control, such as in a chemical plant, or load balancing on a multi-processor

computer system [3].

In nature, an individual within a population competes with one another for resources (e.g. food or water). Besides, an individual within same species often competes with others to attract a mate. Those individuals which are most successful in surviving and attracting mates will have relatively larger number of offspring. On the other hand, poorly performing individuals will produce fewer offspring or may even "die out". This means that the genes from the highly adapted, or "fit" will spread to an increasing number of individuals in each successive generation. In this way, species evolve to become more and more well suited to their environment, see [3].

GA uses a direct analogy of this natural behavior. It works with a population of "individuals", each representing a feasible solution to a given problem. Each individual is assigned a "fitness score" according to how good a solution to the problem is. The highly fit individuals are given opportunities to "reproduce", by "cross breeding" with other individuals in the population. This produces new individuals as "offspring" and the least fit members of the population are less likely to get selected for reproduction, and so "die out". Over many generations, good characteristics are spread throughout the population, being mixed and exchanged with other good characteristics as they go. By favoring the

mating of more fit individuals, the most promising areas of the search space are explored. If GA is designed well, the population will converge to an optimal solution to the problem.

Clearly, the large population of solutions and simultaneously searching for better solutions give the genetic algorithm its power. Indeed, some of the advantages of GA can be summarized as below [3]:

1. Optimizes with continuous or discrete parameters.
2. Does not require derivative information.
3. Simultaneously searches from a wide sampling of the cost surface.
4. Deals with a large number of parameters.
5. Is well suited for parallel computers.
6. Optimizes parameters with extremely complex cost surfaces; they can jump out of a local minimum.
7. Provides a list of optimum parameters, not just a single solution.
8. May encode the parameters so that the optimization is done with the encoded parameters, and
9. Works with numerically generated data, experimental data, or analytical functions.

1.8.1. Constraint Handling Methods in GA

There are three types of method to reduce the number of infeasible solution:

1.8.1.1. Method one – decoder

The concept of this method is to generate gene in a chromosome follows a probability distribution P with bias to the chromosome with better fitness. When P is set to '1', alleles of superclass over-rule the random generation process. Decoder of solution bitstring gives instruction on how to generate feasible physical design for database. These instructions are learned during the adjustment process and it becomes the rules to the generation of feasible solutions. Following the bitstring generation rules, random gene bits' values of '0s' or '1s' are bound to feasible solution. Bitstrings are inspected before infeasible gene bits are generated and therefore adjustment to the solutions is not required. Decoder method randomly generates gene bit if inheritance constraint is not applicable, otherwise it gives instruction to build feasible solution bits in forward and backward direction. Forward decoder stacks up gene bits from front to end while backward decoder does that in the reverse way. Because information of inheritance constraint is learned and re-used for guiding bitstring generation, feasible solutions are generated at lower GA operating cost and

directly evaluated without further repairing process.

1.8.1.2. Method two – reduce infeasible solution fitness

To count the number of violation in a solution, memory is allocated for the indication of inheritance constraints applied to the given inheritance structures. In other words, this is an index for searching inheritance constraint of any given instance variable and all of its correspondences in the inherited classes. Since the inheritances between genes are recognized on such index, we can easily check for and report a chromosome's number of inheritance constraint needed for adjustment.

Constraint handling methods is based on application of special repair algorithms to "correct" any infeasible solutions so generated. Again, such repair algorithms might be computationally intensive to run and the resulting algorithm must be tailored to the particular application. Moreover, for some problems the process of correcting a solution may be as difficult as solving the original problem. The proposed design methodology has major set back which reduce the efficiency in the repair of genotypes with inheritance constraints from the phenotypes. Inheritance constraints lead to mis-representation of genotypes, which require repairing.

1.8.1.3. Method three - penalty function

Additional version of penalty approach is elimination of non-feasible solutions from the population [44]. This technique was used successfully in evolution strategies for numerical optimization problems. The drawback for this approach is that a feasible solution is relatively small and the algorithm spends a significant amount of time evaluating illegal individuals. Moreover, in this approach non-feasible solutions do not contribute to the gene-pool of any population. (important information in infeasible solution is wasted)

1.9. Contributions of this work

Properly designed physical storage structure can facilitate transaction's requests to access to classes in the secondary memory, which is costly in database operation. Physical organization problem that determines whether the instances variables stores in subclass or in superclass of OODB storage structure has been solved with GA by N.Gorla [32].

Gorla devised Edge-to-edge crossover operator and two adjustment methods. They are forward and backward adjustments. Solving the design solution problem with GA is dissatisfactory because of low converging rate and long computational time. This research shows that GA

performance can be improved after using the proposed Enhanced Crossover Operator and customized GA operators.

Gorla's edge-to-edge crossover operator limits crossover operation to the scope of exchanging edge(s) out of selected parents in the solution population to reproduce offspring for the solution pool. Edge crossover bounds the searching space to the combination of edges that are traced back from the initial solution pool. A children solution may be trapped in local optimal in some generation until mutation operators exploit new edge combination to searching space. Poor performance becomes prominent when the edge-to-edge operator handles large number of classes in the hierarchy structure and therefore long computational time is resulted.

With the aim to extend the confined search space of the defined initial population and always maintain the feasible solutions pool, Enhanced Crossover Operator and Enhanced Adjustment Methods (EAMs) for physical storage problem of OODB are introduced and applied for GA in this research. Experiment results reveal that these operators and methods improve the converging rate and reduce computational time in finding a feasible solution. To avoid infeasible solution misrepresenting solutions, it is also important to maintain solution feasibility while the solution pool is converging to the optimal solution. Enhanced

Adjustments Methods consist of forward and backward adjustment, that check inheritance constraints generated in populations while new generation is converging to universal optimal solution. One major contribution of EAM is that it traces inducing inheritance constraints after forward and backward adjustments in the solution children pool. It also ensures solution feasibility during processing each generation, which checks inheritance constraint and corrects infeasible solutions so as to guarantee feasible solution pool. Experimental result depicts that EAM has better capability in handling large class hierarchy and improving the rate of converging to optimal design of feasible solution.

In the same Gorla's university database with 6 classes as an example, the optimal design solution is 32 in the ninth iteration. Gorla's crossover method reaches 35 in the third iteration and 34 in the sixth. Proposed ECO reach 34 in the fifth iteration and reach optimal solution 32 in the seventh. Enhanced Crossover Operator (ECO) and Enhanced Adjustment Method (EAM) has improved the rate of solution convergence while guarantee the solution feasibility in the population. Besides, Enhanced Crossover Operator and EAM can handle database with over 25 classes while maintaining the feasibility of design solutions. This is a significant improvement for OODB that promise the handling of large and complex class hierarchy.

1.10. Outline of this work

This research is organized as follows. In this section, we have discussed the history of object-oriented database development in brief and generally introduced its application. The composition of OODBs and the factors of performance properties, for instance the database access made in class, are depicted in detail so that the readers may understand the motivation of this research. In Chapter 2, we have shown that physical design for the storage of object-oriented database is a combinational optimization problem that is high-complexity in nature. Researchers have developed several physical database models but none of them are universally accepted for the performance design of operating database. After we consider various heuristic methods, using Genetic Algorithm for solving this design problem is the better way that guarantees universal optimal solution. Chapter 3 discusses traditional physical storage models adopted by various Object-Oriented Database Management Systems. After the discussion over transactions operation in the traditional implementation of OODBMS, the evaluation function that measures the performance of transaction over physical storage transaction is explained in details. A university database is used as the illustrative example that depicts the traditional implementation to solve this problem with GA. The improved GA operators and Enhanced Adjustment Methods for better handling of infeasible solutions are discussed in Chapter 4. After presenting the

analytical and empirical results from the experiment, we make the conclusion on improved performance in solving Physical storage problem for OODBs in GA in Chapter 5. Conclusion and future directions are presented in Chapter 6.

Chapter 2

2. Literature Review

2.1. Object-oriented database

Object-oriented databases (OODBs) are known to be rich in functionality but poor in performance. The performance of data access is a major concern in OODBs [35]. One factor in the performance of an object-oriented database is the storage overhead incurred by objects and indices to instances. The issue of physical storage models for OODBs has not been addressed very well in the database literature. Consequently, there are no universally accepted models for the physical storage of the instances in an OODB.

The presented models of OODB structures are not designed with the objective to minimize the number of access in database for transaction

operations, which reduce the transaction processing time of information needed. We focus on the space overhead incurred by the storage model based on the technique used to physically store the attribute values. This issue deals with where the attribute values of the inherited instance variables should be stored. W. Kim [42] presented that the OODB system must determine the physical location of the object quickly by the mapping of the logical identifiers of the objects to their physical addresses. It is obviously important to identify frequent operations in OODBs and to optimize their performance. Properly designed physical database structure improves the database operation performance [25] by minimizing the number of accesses made in the secondary storage when processing transactions. The concept is to store the values of instances in a properly designed physical storage structure in a database so that the number of access by the transaction operations, including update and retrieval, is minimized.

2.2. Object-Oriented Data model

In a basic object-oriented data model, memory is allocated to an object for holding its state. The state of the object is composed of the current values for its instance variables, which are also objects with private memory spaces. In data model, a class is a group of objects that share the same

instance variables so that the storage overhead is reduced. The ISA relation for class organization represents a class hierarchy. In a class hierarchy, a superclass is the class above another class, which is also called subclass. A subclass inherits the instance variables and methods of its superclass. On a logical level, an object-oriented database is arranged as a directed acyclic graph (DAG) representing a class hierarchy (See Figure 3.1 in chapter 3). The logical view of a simple database model represents the logical data concept that instances belong to a class and to all of the superclass of that class.

Object instances are physically stored in the allocated memory in the original logical class (the home class), or additional memory can be allocated for a duplicated copy of object instances in one or more subclasses. Each instance has a unique identifier, which is for the link to the instances being referenced to as the classes. The instances in classes are linked to where the attribute values of inherited instance variables are physically stored with reference to the ISA hierarchy.

The purpose of the classes and the class hierarchy is to avoid the duplicated storage of instance variable names. However, proper arrangement on the duplication of instances being physically stored in the class hierarchy can reduce the number of access to classes which is

anonymous to reducing the operation cost of database.

A universally accepted data model needs to be address properly for the physical storage of the instances in object-oriented databases. Willshire and Kim [24] presented the properties of Physical Storage Models for OODBs and defined the design issues over these models. The performance of these models and analyses the storage cost of these models are examples in the study. N. Gorla [32] based on these physical storage models formulated the evaluation method over various proportions of transaction processing types, which include retrieval and update transactions.

2.3. Physical Storage Model for OODBs

Physical storage structure is one of the major considerations in determination of performance in OODBs, because minimal access to instances is assumed to be synonymous to the reduction of response time. Implementers of object-oriented database management systems use various models of physical structures. Willshire and Kim [24] presented the physical storage models. The strength and weakness of each model were discussed in the database literature but the issue of physical storage of instances in an OODB still was not addressed very well. In the

following section, these models discussed in this section are Home Class (HC), Repeated Class (RC), and Split Instance (SI).

2.3.1. Home Class (HC) Model

HC is used in ORION OODBMS, a prototype database system that manipulated in object-oriented application, presented by J. Banerjee et al. [23], an instance is stored in the lowest subclass in the hierarchy. In Home Class Model, each instance occurs exactly once in the database, thus providing horizontal partitioning of the instances in the database and no duplication of the data. The members of a class are scattered over the class hierarchy represented by DAG. An instance is "pushed" to the lowest level possible in the class hierarchy. All instances contain values of instance variables in the most specific class that they belong with no duplication of data in Home Class. Because the entire instance is contained in exactly one class, horizontal partitioning of the instances in the database is provided for class hierarchy indexing so that all applicable instances are indexed together for the access of user's transaction to the database. This is done to avoid duplication of data storage, but it causes multiple classes to be involved when all the instances in a class are possible targets of retrieval.

Willshire and Kim revealed that HC with no data duplication has the lowest storage overhead at the expense of making retrievals more complicated. When doing retrievals, if all instance are selected in a particular transaction that request the values of all instance variables in the class, all instances of the subclass are to be retrieved until the lowest level of Class Hierarchy is reached. A similar mechanism is employed for deletion and updates of multiple instances. This transaction operation process often involves numerous accesses to levels of subclasses in the secondary memory, which is expensive in the database operation. Once the instance is located, all attribute values of instances are contained in the class and so there is no need to search any further. Owing to an instance and all values of its attributes must be stored in the most specific home class, we consider that HC has better performance in terms of space overhead and retrieval of a single instance that is the case when there is no need to access to the next level of subclass. User's transaction needs often targeted to a few classes for attribute values and some duplication of data that compromised on storage overhead is necessary for the convenience of transaction access. HC do not provide the flexibility to duplicated data storage and make data in the lowest subclasses more complicate to access.

Chu and Ieong discuss attribute partitioning using a transaction-

base approach that designs the attribute partitioning according to the user information need. The difference between various models lies in the manner in which the instance variables and the instances of each class are stored. Since Home Class Model requires all attribute values contain in one class and do not allow for duplication of data, the physical storage of attribute values in an instance could not be allocated to the subclasses based on user's transaction pattern.

In this research, we adopted the data fragmentation, or vertical partitioning, that is used in relational databases to OODBs for performance improvement. We attempted to determine the lowest operating cost of physical storage structure, or data fragmentation, based on user's information needs. For this reason, HC, a horizontal partitioning storage model, is not applied to the design of physical storage structure in OODBs.

2.3.2. Repeated Class (RC) Model

An instance is stored in all the subclasses it applies in Repeated Class Model. If an instance is a member of more than one class, then the data for the instance is stored in each of those classes. Each instance contains all the inherited and home instance variables in its home class and all the subclass it applies. There is an excess of duplicated data that made

data retrieval extremely high performance at the expense of overloading the space overhead. In RC, values of instance variables can be retrieved in one access to class that it applies. In opposite, data update has poor performance. Because instances are duplicated in all applied classes, the values of instance variables stored in these classes need to be accessed until all classes are updated. This is an ideal model for the user's transaction needs that merely for processing retrieval transactions but it is a poor model for update transaction

2.3.3. Split Instance (SI) Model

The Split Instance Model is similar to RC model, except that the inheritance variables are only stored in the superclass not in any subclass. SI model is used in IRIS OODBMS. In SI, an instance is also a member of more than one class but the data for the instance is not inherited or stored in each of those subclasses. In each applicable class, the unique id is stored along with the values of the attributes particular to that class for the instance. Only the id field is duplicated down the ISA hierarchy.

While there is some duplication of data, it is quite minimal in this scheme. Only the unique instance identifier is repeated and is used for forming joins to gather the parts of an instance from the various classes.

The identifier is used to form joins over the classes, so that every class that contains a matching identifier will be accessed for the value of instance variables stored in the original class. There is no significant duplicated data because identifier is used to join classes for the data value. SI has higher performance in data update because all values of instance variables are updated once the original class is completed. In opposite, data retrieval has poor performance. Because values of instance variables only stored in their own classes, the transaction need to access all associated classes until all data are retrieved. This is an ideal model for the user's transaction needs that merely for processing update transactions but it is a poor model for retrieval transactions.

2.4. Solving physical storage design for OODBs.

Gorla [32] presented an object-oriented database design for improved OODB performance. Evaluation Function measures such performance of physical storage design solution based on operating cost that counts the number of access to classes. Because the response time of transaction processing in a database is greatly affected by the number of instance needed to be access in class, the storage structure is designed for a set of frequently used transaction.

The ratio of transaction types, including RETRIEVAL and UPDATE transaction, is one of the major factor that affect the performance of OODBs operation. UPDATE operations are performed in the most inefficient way in RC model. UPDATE operations require visiting more than one class because instance variables may be replicated in several classes. On the other hand, in SI model, the instance variables are not inherited in the subclasses. Thus only home instance variables are stored in each class, thus eliminating redundancy of the instance variables. Because of these update operations are performed more efficiently. However, retrieval operations will be performed in an inefficient way because when processing these operations, it is necessary to visit more than one class to get complete information of the instances. Although RC model is better in general for retrievals compared to SI model, SI performs better than RC if the retrievals are such that they only need home instance variables. The above discussion of RC and SI models can also be done analytically.

2.5. Transaction-Based Approach

Chu and Ieong [41] studied the Transaction-Based Approach to Vertical Partitioning for relational database systems. This approach allows the optimization of the partitioning based on a selected set of important transactions. The object-oriented data model is more complex than the

relational model. Chu and Ieong presented that the number of disk I/O's of all the transaction is the sum of the disk accesses incurred by each transaction. The objective of physical storage model design for OODBs is to reduce the number of disk I/Os in the system, which depends on the transaction access pattern, their access frequencies, and the access methods.

2.6. Minimize database operational cost

A major consideration in determining the performance of OODBs is physical storage structure. N. Gorla [32] presented a methodology in similar manner to the transaction-based approach. The methodology determine OODB storage structure in terms of physical organization of instances in the database that results in the least database operating cost. The determination is as to which instance variables should be inherited and stored in a subclass and which instance variables should be stored in a super class.

Instance variables can be inherited from both their direct superclass and indirect superclass. For example, an instance variable can be inherited by a subclass from its superclass, the instances of which are composed of home variables and inherited variables of these superclasses. The possible number of database designs can be very large even with moderate number

of classes and home variables. Gorla discussed the physical storage problem in searching the minimum database operation cost for OODBs. The problems are described as follows. Given a logical OODB schema and a set of user retrieval and update requests, determine the storage structure that yields the minimum database operating cost.

2.7. Combinational Optimization Method

Gorla proposed a methodology to determine OODB storage structure in which given set of transaction are operated at minimal cost. Instances variables of subclasses are composed of home variables and inherited variables, which may be inherited from both their direct superclass and indirect superclasses. Even a moderate number of classes and instance variables in database schema, the possible number of database designs can be a large population of combination [32]. It is difficult to determine the best, or optimum, solution in a reasonable amount of time even for a small problem. In order to solve this intractable problem, we need an efficient heuristic algorithm that can provide a good solution for larger databases.

2.7.1.1. Heuristics approaches to optimization problems

There are several optimization methods to solve these difficult problems, such as traditional hill climbing and stochastic optimization

techniques such as Simulated Annealing (SA). They present certain disadvantages [46, 47].

In traditional hill climbing, the search process starts from a single point and ignores other potential solution points in the search space. The search is conducted locally from a single point in the solution space. This heuristic search method is easily trapped in local optima. A common practice is to execute a number of runs with different initial solutions, then select the best solution as the final result. Simulated Annealing [33] has been suggested to overcome this problem by using a long cooling schedule to approximate asymptotic convergence. The method [33] is motivated by an analogy to a phenomenon in crystallization. SA [33] is based on the process in which a solid is sufficiently heated to a liquid form followed by cooling, so that the particles arrange themselves into a lattice. By selecting parameters for the initial high temperature and the cooling rate, various solution states are obtained. Unlike hill-climbing techniques, SA allows occasional downhill moves to escape from unattractive local optima. It has been applied to solve the location and the traveling salesman problem.

When compared with a local search method in some facility layout tasks, SA generated better solutions at the expense of prolonged computation time. In each iteration, both techniques search only a single

region defined by the neighborhood of an existing solution. In SA, the attempt to sample different regions of the solution space may only be realized with a cooling schedule that has a high initial temperature, a small temperature decrement in each iteration, and a long annealing chain. All these combine to form a schedule that will demand a prohibitively long CPU time. Most importantly, both SA and local search method do not learn from the solution space already explored; consequently, useful information about the function surface that can be inferred from known solutions is left unused.

GA maintains several solution points in parallel and exploits them to build better solutions thus leading to global optimum. Unlike SA, GA builds better solutions by using information found from already existing good solutions, thus reaching global optimum solution much faster. We therefore choose GA over other optimization algorithms.

2.8. Research in Genetic Algorithm

GAs are becoming popular for their ability to solve complex problems and provide good solutions, not necessarily optimal solutions. The basic idea of the GAs is as follows. A set of candidate solutions in the form of bit-strings, called "chromosomes," is randomly generated. They are the genotypes that are manipulated by the GA. Each of these solutions is evaluated using some performance measure called fitness function. The evaluation routine decodes these structures into some phenotypical structure and assigns a fitness value. Typically, but not necessarily, the chromosomes are bit strings. The two best chromosomes are selected and mated to produce two new chromosomes. This is called crossbreeding, where we crossover the fragments of these solutions to produce offspring from the parents. The offspring are merged with the initial population and the weaker solutions are removed from the solution pool. This becomes the solution pool for the next iteration. Thus, each subsequent generation produces better and better solutions. Mutation involves changing some bits of the bit-string at random. Implementation decision includes variation rates (crossover and mutation), offspring reproduction rate, and population size from the old population.

2.9. Implementation in GA

The generation of solution bitstrings is created as follows. A solution bitstring composed of several substrings. There is one substring corresponding to one superclass-subclass relationship. The substring contains 0s and 1, where a 1 denotes that an instance variable from the superclass is inherited (and stored) in the subclass and a 0 denotes that an instance variable is not stored. Thus, the length of a substring of a superclass-subclass relationship is the number of variables – home or inherited – presented in that superclass.

The pseudo-code of the algorithm is given in Figure 2.1.

```
Set t = 0;
Generate class hierarchy (C-H) of physical storage for OODB;
Generate one set of transaction requests using experiment parameter;
Encode C-H edges to bitstring representation;
Generate randomly initial solution bitstring and checked inheritance
constraint;
Repeat generation process until all feasible solution in P(t);
Evaluate the initial population with the fitness function in P(t);
        While (termination criterion not satisified) do
          t=t+1;
          Reproduction biased fitness selection two parent P1 and P2
          from P(t-1);
          Common edge crossover to selected parent in C(t);
          Recombine solution and mutation in C(t) and store in C'(t);
          Check inheritance constraint and apply forward/backward
          adjustments;
          Evaluate the child solution and rank for fitness in C'(t);
          Replace_select two offspring in C'(t) to P(t);
        End while
```

Figure 2.1: Gorla's GA algorithm

2.10. Fitness function

The design methodology of physical storage structure with the aim to minimize the number of access to classes is based on specific transaction access. The fitness function is simply the number of access to class for transaction. Update operations require visiting more than one class because instance variables may be replicated in several classes.

A fitness function must be used to evaluate the "fitness" of the individuals within the population. Parents are selected from the population using a scheme that favors the more fit individuals to produce offspring. Parent selection is a process that allocates reproductive opportunities to individuals. Good individuals will probably have more opportunities to be selected as parents and poor ones may not be at all. The biased selection enables the convergence of the search. As the population converges, so the range of fitness in the population reduces. However, this sometimes leads to premature convergence and slow finishing. A few comparatively highly fit individuals may rapidly come to dominate the population, so as to converge on a local optimum, which is also called premature convergence. After many generations, the population will have largely converged, but may still not have precisely located the global optimum. The average

fitness will be high, but there may be little difference between the best and the average individuals. Consequently, the fitness function create gradients between values are too close to push the generation towards the global optimal solution. The commonly employed methods include fitness scaling and fitness ranking. In the implementation stage, we applied fitness ranking method to ensure that selection gradient push the range of fitness in population to be converged.

2.11. Crossover operation

The intuitive idea behind crossover: given two individuals who are highly fit, but for different reasons, ideally what we would like to do is create a new individual that combines the best features from each. Since we do not know which features account for the good performance, the best we can do is to recombine features at random. This is how crossover operates. It treats these features as building blocks scattered throughout the population and tries to recombine them into better individuals via crossover. Goldberg [6, 9] reveals that better-fit schemata in the solution are preserved in new population and converging to optimal solution after many generations. The best-known alternative to one- and two-point crossover is uniform crossover. Uniform crossover randomly swaps individual bits between the two parents.

2.12. Encoding and Representation

Although GAs typically uses a bitstring representation, GAs is not restricted to bitstrings. A number of early proponents of GAs uses other representations, such as real-valued parameters, permutations, and treelike hierarchies. Koza's genetic programming paradigm is a GA-based method for evolving programs, where the data structures are LISP S-expressions, and crossover creates new LISP S-expressions by exchanging subtrees from the two parents.

Traditionally, GA uses binary representation, which is often termed chromosome. However, since each digit has cardinality of 2, higher cardinality alphabets have been used and some researchers claim that it has advantages over the traditional coding. Several non-binary coding methods were proposed such as adjacency, ordinal, path and ordered. In our research, 1/0 Integer programming is used to represent the inheritance of instance variable in the given physical storage structure. We applied binary bitstring representation to solve this combination optimization problem.

2.13. Parent Selection in Crossover Operation

Parent selection is a process that allocates reproductive opportunities to individuals. The biased selection enables the convergence of the search. As the population converges, so the range of fitness in the population reduces. However, this sometimes leads to premature convergence and slow finishing.

Premature convergence means that the genes from a few comparatively highly fit (but not optimal) individuals may rapidly come to dominate the population, causing it to converge on a local optimum. Slow finishing is the reverse problem to premature convergence. After many generations, the population will have largely converged, but may still not have precisely located the global optimum. The average fitness will be high, and there may be litter difference between the best and the average individuals. Consequently there is an insufficient gradient in the fitness function to push the GA towards the global optimal solution.

There are many methods to overcome these problems. Several are described in [45]. The commonly employed methods include fitness scaling and fitness ranking. Fitness ranking overcomes the reliance on an extreme individual. Individuals are sorted in order of raw fitness, and then

reproductive values are assigned according to rank. In fitness scaling, the maximum numbers of reproductive values are assigned according to rank. Whitley [14] conducts some experiments and shows that fitness ranking to be superior to fitness scaling.

2.14. Reproductive selection

Whitley's GENITOR [14] creates one child each cycle, selecting the parents using ranked selection, and then replacing the worst member of the population with the new child [14]. Syswerda's steady-state GA [38] creates two children each cycle, selecting parents using ranked selection, and then stochastically choosing two individuals to be replaced, with a bias towards the worst individuals in the parent population [39]. Eshelman's CHC [15] uses unbiased reproductive selection by randomly pairing all the members of the parent population, and then replacing the worst individuals of the parent population with the better individuals of the child population.

In our implementation, we adopt Eshelman's CHC [15] that randomly pairing all the members of parent population, i.e. in a population with 20 members, 10 pairs of parents reproduces 10 kids to the new population pool.

2.14.1. Selection of Crossover Operator

In the case of combinatorial problems such as the traveling salesman problem, a number of order-based or sequencing crossover operators have been proposed. The choice of operator will depend upon one's goal. If the goal is to solve a TSP, then preserving adjacency information will be the priority, which suggests a crossover operator that operates on common edges [46]. On the other hand, if the goal is to solve a scheduling problem, then preserving relative order is likely to be the priority, which suggests an order preserving crossover operator. For example, chooses several positions at random in the first parent, and then produces a child so that the relative order of the chosen elements in the first parent is imposed upon the second parent.

2.14.2. Replacement

There are two replacement approaches, named generation gap and steady-state replacement. The generation gap is defined as the proportion of individuals in the population, which are replaced in each generation. Most work has used a generation gap of 1, i.e. the whole population is replaced in each generation [15]. However, a more recent trend has favored steady-state replacement [38]. It replaces a few individuals in each generation.

In our implementation, we adopt generation gap of 0.8, i.e. in each generation with 20 chromosomes only 4 will survive in the old population.

2.15. The Use of Constraint Handling Method

A genetic algorithm generates a sequence of parameters to be tested using the system model, objective function, and the constraints. Typical approach is to evaluate the objective function, and check to see if any constraints are violated. If not, the parameter set is assigned the fitness value corresponding to the objective function evaluation. If constraints are violated, the solution is infeasible and thus has no fitness. This procedure is fine except that many practical problems are highly constrained; finding a feasible point is almost as difficult as finding the best.

Researchers devise constraint handling methods in the categories of penalty approach, application of adjustment (repairing) method, penalty function for infeasible solution, and decoders a special representation mapping [44].

There are three types of method to reduce the number of infeasible solution [44]:

2.15.1. Penalty function

Sometimes it is possible to extend the domain of function $EVAL_f$, evaluation function for feasible solution, to handle infeasible individuals, EVAL u(x) = $EVAL_f$ f(x) +/- Q(x), where Q(x) represent either a penalty for infeasible individual x, or a cost for repairing such an individual.

Penalty function is common in combinational optimization problem for handling infeasible solution. Additional cost incurred by the penalty function made infeasible solutions fail the selection for replacement to the solution pool over evolution.

To impose penalty to a function, the presence of infeasible solution in solution population is perhaps by degrading their fitness ranking in relation to the degree of constraint violation. This is what is done in a penalty method.

Because the fitness of infeasible solution is invalid for mating

selection and solution replacement, computational resources for random generation and genetic altering processes in certain extent are abandoned subject to the subsequent handling method. Some researchers suggest death penalty to infeasible solution, or the elimination of all infeasible solution from the initial and offspring solution pool, so that no constraint handling method is required. Previous researchers have identified that suggestion is impossible for some problem domain in searching space with large infeasible set of solution. Measuring that in terms of computational resources, searching for feasible solutions in such search space is as difficult as local searching method for optimal solution. Death penalty is not an efficiency constraint handling method for handling large infeasible solutions set of inheritance constraint.

2.15.2. Decoder gives instruction to build feasible solution

Decoder offers an interesting option for all practitioners of evolutionary techniques. In these techniques a chromosome "gives instructions" on how to build a feasible solution. For example, a sequence of items for the knapsack problem can be interpreted as: "take an item if possible" – such interpretation would lead always to feasible solutions.

2.15.3. Adjustment method

It is relatively easy to 'repair' an infeasible individual. Such a repaired version can be used either for evaluation only, where x is a repaired version of y, or it can also replace the original individual in the population. Note that the repaired version of solution 'm' might be the optimum 'X'.

Gorla's [32] approach applies inheritance constraint to check if the generated bitstring is invalid solution. One or more gene, which represented instance variable's superclass-subclass relationship, may violate the inheritance constraint in a candidate bitstring solution. Repairing infeasible solution for replacement is a traditional approach to handle the invalid genes in the solution. After invalid genes are adjusted by the violation handling process, better fitness solution becomes the candidate for replacement in the population of solution. Adjustment operation repairs the solution which is logically infeasible, which is similar to the relationship that children without parent. Gorla [32] proposed two-adjustment method: Forward Adjustment and Backward Adjustment. It has not been discussed the adjustment method in detail for initial candidate solution pool although inheritance constraint is applied to check this solution pool.

The process of repairing infeasible individuals is related to a combination of learning and evolution. Learning and evolution interact with each other: the fitness value of the improvement is transferred to the individual. In that way a local search is analogous to learning that occurs during one generation of a particular string.

The weakness of these methods is in their problem dependence. For each particular problem a specific repair algorithm should be designed. Moreover, there are no standard heuristics on design of such algorithms: usually it is possible to use a greedy repair, random repair, or any other heuristic, which would guide the repair process. Also, for some problems the process of repairing infeasible individuals might be as complex as solving the original problem.

Chapter 3

3. Solving Physical Storage Problem for OODB using GA

A methodology was developed to determine OODB storage structure. With the aim to meet users' database requirements in the most economical manner, physical design for OODB storage is to determine either an instance variable should be inherited and stored in a subclass or the otherwise in a superclass. Instance variables can be inherited from both their direct superclass and indirect superclass. The latter is an instance variable can be inherited by a subclass from its superclasses.

3.1. Physical storage models for OODB

Physical design for database is an important factor to the operation performance of object-oriented database (OODB). In OODB literature, researchers proposed several physical storage models [24] for the instances storing in classes, such as Repeated Class and Split Instances [25]. Some models are implemented by the object-oriented database management system (OODBMS) available in the market, such as Iris [11] and ORION [40]; However, none of these models are universally accepted for the database storage design.

Gorla [32] solved the physical storage design problem with GA. The university database example is adopted from Gorla as an illustrative example on the physical organization OODB. Applied our enhanced GA operators for solving identical database design problem, we can demonstrate improved performance over traditional GA operators.

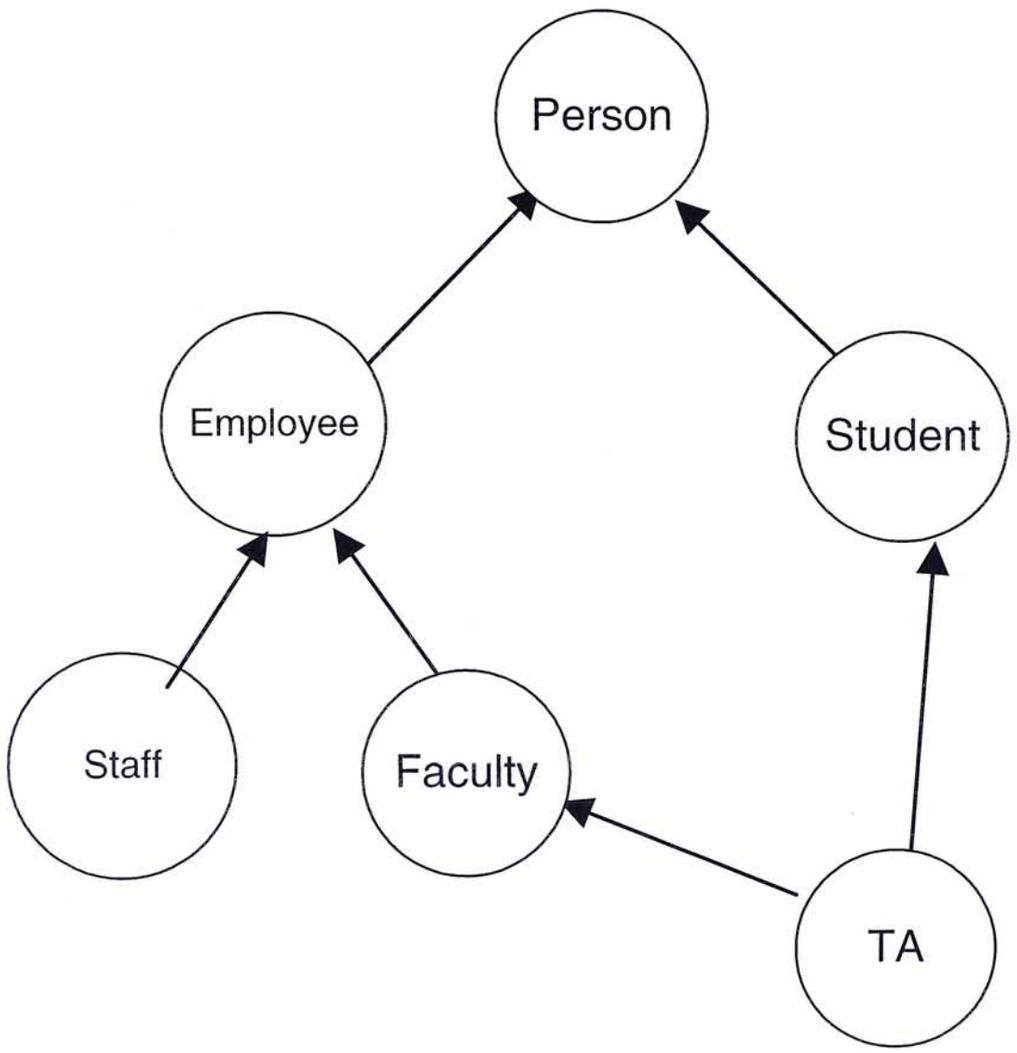


Figure 3.1: Class-Hierarchy structure of a University database.

3.2. Database operation for transactions

The concept of this design optimization problem is to design a physical storage structure that minimize the number of accesses needed in the database to satisfy the transaction requests, which consists of RETRIEVAL

and UPDATE transaction operations. Based on the transaction-based approach to vertical partitioning in relational database design [41], cost of database operations are formulated in similar manner applied to classes in OODB. This evaluation function serves two purposes: a) understanding the operation of database how transaction process access to instances in classes, and b) evaluating database operating cost by counting the number of database accesses until all transaction requests are satisfied.

To understand how transactions are processed in database operation, properties of different transaction types are to be discussed. Transaction's requests can be satisfied if the needed data is stored in the class. Operating cost of a given set of transaction requests to access the database is depends on where the object instances are stored in physical storage. Design solution is encoded in bitstring representation. Bitstring is applied to design solution using which to determinate of physical storage structure of instance variables in superclass or subclass.

Class: EMPLOYEE

Instance variable:	**SSN**	**Name**	**Age**	**Date-Hired**	**Salary**
Instance #1	234	John	30	5/20/89	32,000
Instance #2	450	Nara	29	6/05/90	15,000
Instance #3	498	Swarna	25	6/01/92	12,000
Instance #4	775	Mary	36	12/01/85	65,000
Instance #5	792	Robin	42	01/01/80	59,000
Instance #6	802	Bill	24	01/09/95	42,000

Table 3.1: Repeated Class Model for instance object stored in EMPLOYEE class

Class: EMPLOYEE

Instance variable:	**SSN**	**Date-Hired**	**Salary**
Instance #1	234	5/20/89	32,000
Instance #2	450	6/05/90	15,000
Instance #3	498	6/01/92	12,000
Instance #4	775	12/01/85	65,000
Instance #5	792	01/01/80	59,000
Instance #6	802	01/09/95	42,000

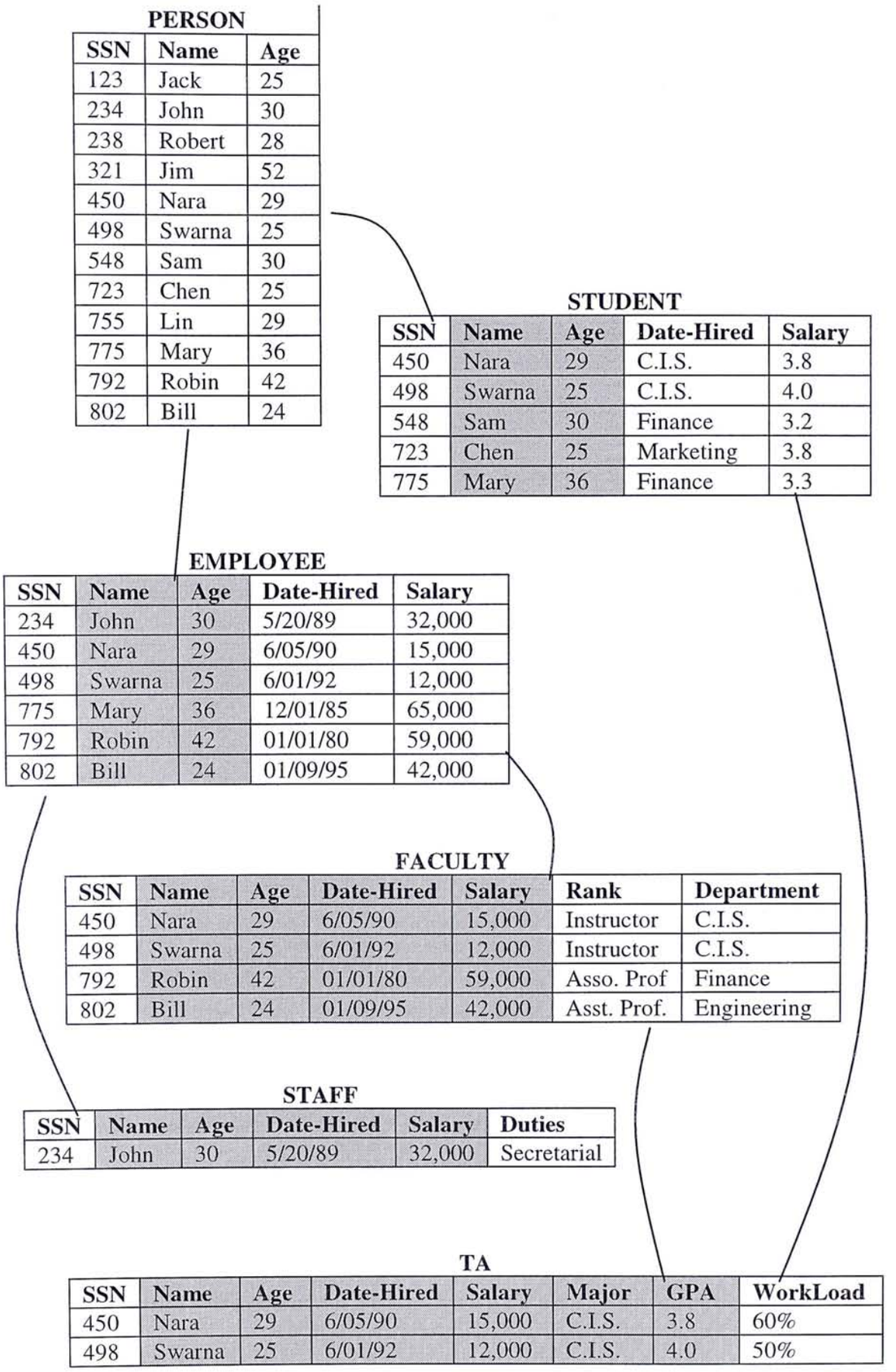
Table 3.2: Split Instance Model for instance object stored in EMPLOYEE class

This design optimization problem is introduced because RETRIEVAL and UPDATE transactions behave differently. Split Instance (SI) model and Repeated Class (RC) model are devised by the researchers [24]. In the Split Instance (SI) model, the instance variables are not inherited (and stored) in the subclasses; In RC model, an instance is stored in all the subclasses it applies. RC is favor to retrieval transaction while SI is favor to UPDATE transaction. RETRIEVAL transaction prefers data to repeatedly stored in all classes so that all instance is retrieved in one access to class; UPDATE transaction prefers data to merely stored in super class so that once the home instance is updated all instance in subclass follows. For the university database example in Figure 3.2, in Repeated Class (RC) Model a RETRIEVAL transaction can obtain instance object SSN 450 and associated instance variable 'Age' from the class of **TA** in one access. A UPDATE transaction would repeatedly access to **STUDENT**, **EMPLOYEE**, **FACULTY**, and **PERSON** for updating the instance variable 'Age' stored in instance SSN 450.

The second property of transaction is that UPDATE transaction require two times of access to a single instance variable instead of one time access in retrieval transaction. In updating operation, an instance variable stored in the class is removed from the memory in the first access,

and then new value is stored in the second access. Retrieval transaction access to the class once because it obtains the instance variable without modification to the data. Because of these properties of transaction processing, the aim of this research is to develop a design methodology that determine the best physical storage solution to serves any given set of transaction with various proportion of RETRIEVAL and UPDATE transactions.

Database access to instance is one of the major database operating costs because some instance variable's data stored distance away is frequently requested in transactions. If the physical design for database is not efficient, such data search continue to made access in classes until it obtains from where the data physically stored. This process occupied computational resources and reduced database efficiency. An efficient design of physical storage solution not only incurs minimal database operating cost in the database but also satisfies all transaction processes to access classes.



PERSON

SSN	Name	Age
123	Jack	25
234	John	30
238	Robert	28
321	Jim	52
450	Nara	29
498	Swarna	25
548	Sam	30
723	Chen	25
755	Lin	29
775	Mary	36
792	Robin	42
802	Bill	24

STUDENT

SSN	Name	Age	Date-Hired	Salary
450	Nara	29	C.I.S.	3.8
498	Swarna	25	C.I.S.	4.0
548	Sam	30	Finance	3.2
723	Chen	25	Marketing	3.8
775	Mary	36	Finance	3.3

EMPLOYEE

SSN	Name	Age	Date-Hired	Salary
234	John	30	5/20/89	32,000
450	Nara	29	6/05/90	15,000
498	Swarna	25	6/01/92	12,000
775	Mary	36	12/01/85	65,000
792	Robin	42	01/01/80	59,000
802	Bill	24	01/09/95	42,000

FACULTY

SSN	Name	Age	Date-Hired	Salary	Rank	Department
450	Nara	29	6/05/90	15,000	Instructor	C.I.S.
498	Swarna	25	6/01/92	12,000	Instructor	C.I.S.
792	Robin	42	01/01/80	59,000	Asso. Prof	Finance
802	Bill	24	01/09/95	42,000	Asst. Prof.	Engineering

STAFF

SSN	Name	Age	Date-Hired	Salary	Duties
234	John	30	5/20/89	32,000	Secretarial

TA

SSN	Name	Age	Date-Hired	Salary	Major	GPA	WorkLoad
450	Nara	29	6/05/90	15,000	C.I.S.	3.8	60%
498	Swarna	25	6/01/92	12,000	C.I.S.	4.0	50%

Figure 3.2: Repeated class model

3.3. Properly designed physical storage structure

Proper physical storage structure is designed to suit transaction request's access pattern during database operation. Gorla formulated this design problem as a combinational optimization problem and depicted its high-complexity in determination of optimal solution. This design problem primarily determines whether an instance variable should be stored in the superclass or in the subclass for transaction to access. Minimal number of access to classes when processing transaction's request for data significantly reduce database operation in the secondary memory, which is costly to database performance. Gorla solves the physical storage problem with Genetic Algorithm for any given set of transaction.

Solving this problem with GA has better performance than heuristic methods. Different from other heuristic methods, such as Simulated Annealing, solving problem in GA is not to be trapped in local optimal solution but guarantee universal optimal solution. Genetic Algorithm has been an independent research topic over for decades. GA is a powerful tool for solving optimization problem because its GA operators and constraint handling methods can be specifically designed for tackling particular problem. This research aims to improve the GA performance for

solving this physical storage problem. Various GA operators and inheritance constraint handling methods are proposed in this research and those operators in experiment are tested.

3.4. Fitness Evaluation

The objective of design of physical storage structure for OODBs is to satisfy user's request of RETRIEVAL and UPDATE transactions in the most economical manner. Cost function is formulated to count the number of accesses to classes until all instance variables are obtained as requested by the transaction. Class objects and the associated instance variables are stored in physical storage memory of corresponding classes. For solving single objective optimization problem with GA, cost function that measures the number of data access to database equal to the fitness function for evaluation of solution's performance.

Fitness evaluation function is the only link between physical design and GA. It is important because the function reflect the goodness of database design which drives the search to converge to optimal solution. Based on the evaluation concept of this function, a poor design solution allows transaction to repeatedly access to more inherited classes for instances while the minimal design properly stored the instance in classes

close enough for most transactions to access.

In this research, we briefly discuss how Gorla's evaluation function is applied to design solution for database operating performance evaluation. Researchers consider total number of database accesses made in the classes as a major operating cost because this process requires access to the secondary memory that is computationally expensive. Based on this concept, database-operating cost is formulated in 1-0 integer programming for evaluating the performance of solution. Using this evaluation function, Gorla solved this design problem in GA as a minimal operation cost optimization problem. The objective of this function is minimizing the total number of database accesses made in the classes for processing transactions' requests.

$$\text{Min} \quad f(t) = \sum C_t \sum_t Q_{tj_j} * T_{tj}$$

$$s/t \quad T_{tj} \le T_{ti}, \text{ where } i \text{ is a Superclass of } j, \ \forall ij$$

C_t: cost coefficient for the transaction types of RETRIVEAL (C_t=1) or UPDATE (C_t=2)
Q_{tj}: determines which transaction t needs to access class j
T_{tj}: number of instances in class j to be accessed by transaction t

Figure 3.3: Objective function

If the condition is satisfied, the amount of object instances $\mathbf{T_{tj}}$ in class **j** is the number of accesses to this class **j** for the needed instance variable. An instance variable **k** presents in class **j** either which is inherited from its subclass or is a home class variable. Successfully processed transactions obtain the needed instance variables from the instances in class. Instance variables that not successfully processed will continue to check for data presence in the inherited classes until the inheritance reaches home class where the original data is stored. The next operation will process how many instances are needed to be accessed for the instance variable. This process counts the amount of object instances $\mathbf{T_{tj}}$ in class. After finish these processes, the transaction type has to be checked. For UPDATE transaction, the total number of accesses to instances double the access cost, where the multiplier $\mathbf{C_t} = 2$. Referring to the earlier discussion, UPDATE transaction removes existing instance variable in the first access and replace new data to the instance in the second. These processes continue until all transaction requests are completed. The sum of access cost to process the transactions is the fitness of design solution.

3.5. Initial population

Traditional GA approach randomly generates alleles for individual chromosome and stores in the initial bitstring population. Inheritance constraint is applied to each chromosome for checking, and invalid chromosomes are removed from the initial population. Infeasible solutions are penalized by reducing their fitness, or the solution has more chance to die out in the next generation. This initial chromosome generation process is repeated until the initial population is filled by valid solutions. For problem with small feasible solution set, this initial population can substantially reduce GA performance.

3.6. Cross-breeding

Cross-breeding is composed of solution crossover and mutation operations. Crossover creates a new individual that combines the best features from each parent. Parent selection process is biased to high-fitness solutions from the pool and replacement selection is similar but biased to poor-fitness solution in the old population.

Operational mechanism of cross-breeding is to add higher fitness solutions to the solution pool and remove equal number of inferior

solutions from the pool based on selection factors. Syswerda's [38] steady-state GA creates two individuals to be replaced, with a bias towards the worst individual in the parent population. A pair of parent is selected from the population size to reproduce a pair of children. After evaluation for solution cost and performance fitness, the better children will replace same number of worse solutions from old population. Crossover position on the chromosome is a random selection process. Better-fit schemata that contains in chromosome segment in the solution are preserved in new population and converging to optimal solution after many generations. Bitstring solution is composed of two or more sections of edge substrings. Each parent randomly contributes one or more consecutive edge(s) to insert to the corresponding edge position of each other.

Mutation operation is done by selecting one bit at random and deciding to flip the bit or not, by using random number. Its function can exploit searching space contains solution that has not been explored. Mutated children solutions are also checked with inheritance constraint for subsequently constraint handling, if necessary.

Cross-breeding process continue counting the number of generation until the termination condition is satisfied. This process is repeated until the crossed-over solutions are worse than the parent

solutions, or stop after a certain number of generations from the pool is considered the desired solution.

3.7. GA Operators

The concept of GA is that combining elements of superior fitness solutions and then using these solutions to replace the inferior fitness ones will lead to solutions with even higher fitness. In Gorla's offspring solution replacement operation, inferior fitness solutions in the population are replaced by equal number of better-fitness offspring solutions. Two offspring solution bitstrings are evaluated by objective function for database operating cost, fitness of solution, and selection factor. These results of offspring solutions determine whether the inferior solutions in population are sustained or to be replaced by the offspring. Finally, each string in the new population is mutated. This is usually achieved by a very low probability. Mutation preserves diversity in the population and allows for a wider exploration of the random searching space. Genetic operations are repeated until the crossed-over solutions are worse than the parent solution or alternatively the algorithm may be stopped after a certain time is elapsed and the best solution from the pool is considered the desired solution.

3.8. Physical Design Problem Formulation for GA

Firstly, we implement Gorla's genetic algorithm method to solve the optimization problem P: *Minmize* $f(x)$, $x \in X$ Function $f(x)$ has been discussed in this Chapter. Suppose that the fitness function is $f(x)$, and that a solution x^i is encoded in a string s^i of length M. In generation (iteration) t, we distinguish between the set of solutions $\{x^i\}$ *and* X^t , and the set of strings or chromosomes $\{s^i\} = S^i$ from which the mating population R^i is selected. Select the (even) population size N, the probability of crossover a, and the probability of mutation r. As a stop criterion, we will terminate the algorithm after T generations have been performed. The algorithm is initialized by defining the set of N chromosomes S^i, the corresponding N solutions X^i and setting $t:= 0$. Find the best solution $x^* \in X^0$ that gives the lowest objective function value z^*, and save x^* *and* z^*.

3.9. Representation and Encoding

Generation of solution bitstrings: A solution bitstring composed of several substrings. There is one substring corresponding to one superclass-subclass relationship. The substring contains 0s and 1s, where a 1 denotes that an instance variable from the superclass is inherited (and stored) in the subclass and a 0 denotes that an instance variable is not stored. Thus, the

length of a substring of a superclass-subclass relationship is the number of variables – home and inherited – present in that superclass.

3.10. Solving Physical Storage Problem for OODB in GA

3.10.1. Representation of design solution

We briefly discuss Gorla's methodology [32] for the OODBs design problem in GA. Each solution is encoded in form of bitstring to represent a design solution. These strings are often called chromosomes, and their entries genes, further emphasizing the genetic analogy. As shown in Figure 3.4, the same university database from Gorla is used as an illustrative example that helps to depict the database design problem.

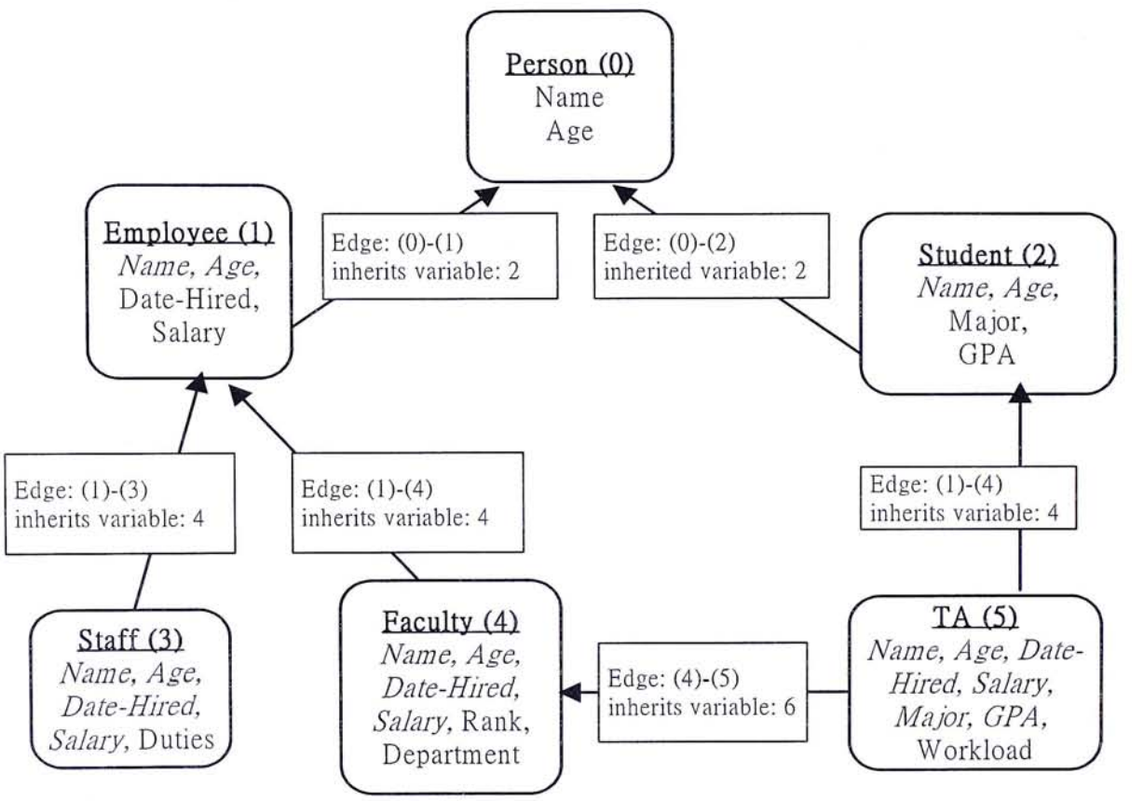


Figure 3.4: Class Hierarchy of University Database

Class Hierarchy is represented by directed acyclic graph (DAG) in which a class may have more than one superclass, as depicted in Figure 3.4. In this University database example, superclass - subclass edges integrate the model of class inheritance relationship, or the class hierarchy. The design of physical storage structure is represented by six substring edges. Besides, the length of substring consisting of solution bits

represents the number of instance variables to be inherited in the subclasses. Consider a subclass **STAFF** concerning all employee in the company. Class hierarchy is essentially table inheritance. Subclass is the specification of superclass; in the otherwise, superclass is the generalization of subclass.

3.10.2. Encoding

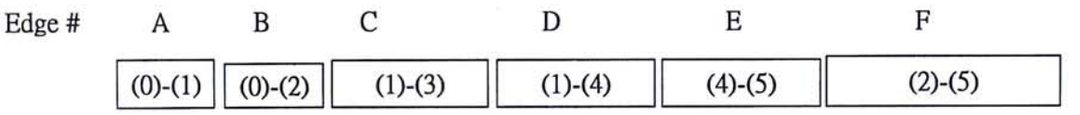


Figure 3.5: Encoded edge representation in bitstring format

This Class Hierarchy is encoded to six substring edges and each edge represents the physical inheritance structure of instance variables superclass to subclass, as shown in Figure 3.5. All solution substrings composed of solution bits that contain 1's and 0's. This binary representation determines whether an instance variable should be inherited to the subclass or stored in the superclass in the physical design. In the example, Gorla's implementation uses a population size of 20 bitstrings and 22 bits in each string. For example, edge (0)-(1) represent class

inheritance from superclass **PERSON** (0) to subclass **EMPLOYEE** (1). A '1' signifies that an instance variable is inherited and stored in the subclass and a '0' imply that it is not inherited and stored in the subclass. A '10' contained in the edge (0)-(1) represents **PERSON** instance variable 'Name' is inherited but 'Age' is not inherited to the **EMPLOYEE**. A solution string '00 00 0000 0000 0000 000000' represents SI model, since none of the instance variables are inherited. Similarly, a solution string '11 11 1111 1111 1111 111111' represents RC model, since all the instance variables are inherited from each class. Since **TA** involves multiple inheritance, **TA** can inherit Name and Age either through **STUDENT** or through **FACULTY** (not necessarily both).

The edge on the graph	Possible variable inheritance	Example substring
1. PERSON → EMPLOYEE	Name, Age	10
2. PERSON → STUDENT	Name, Age	00
3. EMPLOYEE → STAFF	Name, Age, Date-Hired, Salary	1011
4. EMPLOYEE → FACULTY	Name, Age, Date-Hired, Salary	1001
5. STUDENT → TA	Name, Age, Major, GPA	0011
6. FACULTY → TA	Name, Age, Date-Hired, Salary, Rank, Dept	100110

Table 3.3: The edges and their corresponding example substring.

3.10.3. Initial population

Several solution bistrings are randomly generated based on the solution representation format and the number. We apply the inheritance constraint to each bitstring to check if the bitstring is a valid one. Infeasible solution is removed from the initial population and random generation is repeated until the population is filled up with feasible solutions.

3.10.4. Parent Selection for breeding

Population of mating or reproducing solutions is selected using a probability distribution that favors those solutions with better objective values, or higher fitness. In Gorla's approach, objective function is equivalent to the fitness evaluation that measure database-operating cost on transactions. This evaluation function has been discussed in earlier section. Using objective function in Figure 3.3 to evaluate the Fitness-of-solution of each design solution in the population. Results of this that represent solution's relative fitness in the pool are scaled for the computation to selection-factor-of-solution. These equations are shown in Figure 3.6.

$$Fitness - of - solution_i = 1 - Solution\ \cos t_i \,/\, \sum Solution\ \cos t_i$$

$$Selection\text{-}factor\text{-}of\text{-}solution_i = Fitness\text{-}of\text{-}solution_i \,/\, \sum Fitness\text{-}of\text{-}solution_i$$

where i is the number of bitstrings

Figure 3.6: Fitness function and selection factor for each solution bitstring.

Using roulette wheel mechanism, fitness of solution is linearly scaled to a selection factor, which is the probability distribution for mating selection. Once the mating population has been selected, pairs of string are subjected to the crossover operator that shuffles the parent genes into two new solutions. Consider a point at random in the bitstrings and obtain two offsprings by crossing over. Gorla's crossover operation considered one randomly selected edge from the parent bitstring for cross-breeding as shown in Figure 3.7.

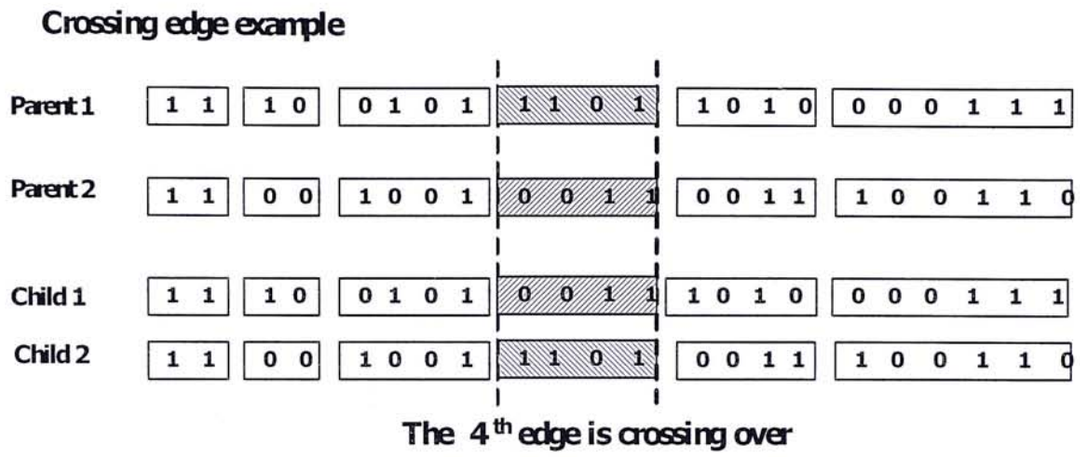


Figure 3.7: Crossing edge example.

Gorla's edge-to-edge crossover operator considers crossing one or more edges of parent genes to reproduce offspring solutions. Edge substring's inheritance information is preserved and inherited to their offspring after checking inheritance constraint is valid. This operation is done by randomly select one or more edges and then swapping the selected parent edge segments to reproduce equal number of offspring solutions. Offspring solutions are derived from these parent solutions. A child or offspring contains copies of the genetic material of the parents, but it has been rearranged, so that represents a different solution to the optimization problem. In the above Figure 3.7, the fourth edge is selected for crossing over. The fourth edge '1101' in parent 1 rearranged with the corresponding position edge '0011' in parent 2. Reproduced Child 1 and

Child 2, which represent different solutions to the physical organization problem, are evaluated by function of database-operating cost for solution performance.

3.11. Traditional Constraint handling method

In Gorla's implementation, bitstring generation, crossover position selection and mutation selection are all random processes. As we wish the better solution segments are concatenated while maintain scatter random search, these genetic operations may induce incorrect and incomplete representation to the design solution. These solutions are adjusted in someway assure that they are feasible solution before the evaluation of fitness.

Solution adjustment method is a problem-specific constraint handling method. It is designed to tackle with infeasible solution in the solution and correct misrepresentation sectors. Gorla's adjustment method repair an infeasible solution in two ways that result in two candidate solutions for the selection to offspring population. Two repairing solutions are generated after forward adjustment and backward adjustment. With Gorla's adjustment method, infeasible solutions are stored in a temporary storage for adjustment. Solutions with better fitness are selected back to

the pool of feasible solution. These adjustment methods are basically repairing methods that amend one-side of invalid edge to cope with its associated edge on the other-side.

Gorla's proposed forward adjustment assumes the first segment is correct and modifications are made to the second segment so that inheritance constraint is satisfied. Backward adjustment assumes the second segment is correct and changes are made to the first segment so that inheritance constraints are satisfied. In the Figure 3.8, Child 1 is checked a violated bitstring after the fourth common edge is crossing. Instance variables that represented by the fourth allele bit '0' in edge four are source of violation. Child 2 is checked but not violated inheritance constraint. We need to repair Child 1 before evaluation of fitness. Forward Adjustment method assume instance variable in common edge four is correct while instance variables being inherited from edge four to edge six is subject to adjustment. The result of forward adjustment is shown in Forward Adjusted Child 1 in which the fourth instance variable is adjusted to '0' that is highlighted. In similar manner, Backward Adjustment method assume instance variable in common edge four is correct while instance variable at second allele of edge one is subject to adjustment. The result of backward adjustment is shown in Backward Adjusted Child 1 in which the second instance variable is adjusted to '1'

that is highlighted.

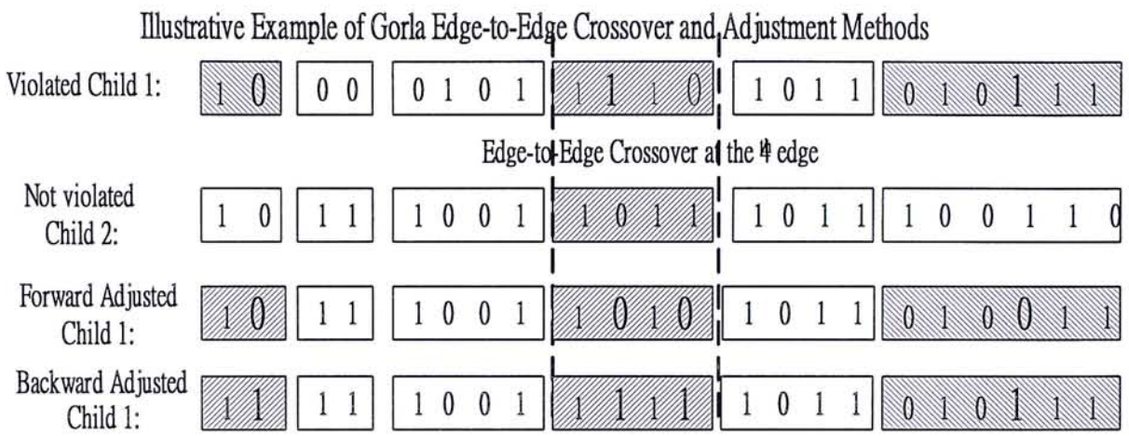


Figure 3.8: Illustrative example of Gorla's Edge-to-Edge Crossover and Adjustment Methods

3.11.1. Improve the Performance of Inheritance Constraint Handling methods

Two repaired solutions are evaluated and solutions with higher fitness can replace solutions in the mating pool. Adjustment method is more efficient in handling constraints because infeasible solutions set are to be amended to become an element of feasible solution set. This method reduces computational time in searching if the feasible solution set is relatively scattered.

In the university database example, an initial chromosome population as the solution seed is randomly generated for solving this optimization problem in Genetic Algorithm. Anyone bit of the solution violated inheritance constraint is an infeasible solution. A checked infeasible solution is invalid for evaluating fitness. Chromosomes in the solution population and offspring solutions in the pool are checked for inheritance validity, or otherwise they are adjusted before going on to the next generation. To handling this kind of solution, Gorla devise forward and backward adjustments where each infeasible solution generates two corrected solutions. Since these adjustment approaches are computational costly for every solution the pool, there are space for improvements. Gorla's adjustment methods and our propose Enhance Adjustment Methods are compared in the next chapter.

3.12. Weakness in Gorla's GA approach

Gorla's methodology for repairing solution is poor in time performance. The average time taken for GA to solve a database problem with five classes is 1.4 min, while solving the problem for the schema with 25 classes is 60 min in 100 generations. Because of this computational limitation, he recommends to use of this methodology for database with less than 25 classes, or to bind the range up to 20 iterations then stop for better but not optimal solutions. To improve GA performance for solving the problem, we propose Enhanced Crossover Operator and Enhanced Adjustment Methods for solving physical storage problem with GA.

Chapter 4

4. Proposed Methodology

We illustrate our design method using the University-database example. The following steps for the OODB design with Genetic Algorithm. To solve this physical organization problem with GA, Class-Hierarchy is encoded to genotype representation in form of bitstring. We have applied our enhanced crossover operator and propagation adjustment methods for solving identical set of transaction's requests over the Gorla's university database example. Enhanced Genetic Algorithm for solving OODB physical storage problem is shown in Figure 4.1.

Set t = 0;
Generate OODB class hierarchy(C-H) storage structure using experiment parameter;
Generate one set of transaction requests using experiment parameter;
Encode C-H edges to bitstring representation;
Generate randomly initial solution bitstring;
Apply Enhance Adjustment Method to solution that violate inheritance constraint;
Better-fit solution x^i store in the initial population P(t) := X^0
Use objective function to evaluate bitstring for fitness of x^i in P(t);

While (termination criterion not satisfied) do
t=t+1;
Using unbiased reproduction selection to pairing up all P1 and P2 in P(t-1);
Using Enchanced Crossover to all parent in C(t);
Recombine solution in C(t) and store in C'(t);
If (any child solution in the kid solution pool violates inheritance constraint)
Enhanced Adjustment Methods create two adjusted solutions to C'(t);
Mutated kids are checked and adjusted in C'(t);
Evaluate the child solution and rank that by fitness in C'(t);
Better-fit parent solutions survive in P(t), subject to the rate of survival;
Replacement select better fit child in C'(t) to P(t) until P(t) is filled;
End if
End while

Figure 4.1: GA Algorithm

Gorla's original algorithm for solving the same problem is adopted

for the baseline measure on GA performance. Results reveal that Enhanced GA operators are outperformed in two major performances. Using our enhanced GA operators, solution population converges to the optimal solution of 32 in generation range from 4 to 12 after 10 trials. Compared to that of Gorla's GA performance, our algorithm's converging rate is much faster than that of 9 trials to reach near optimal solution 34. Computation time for solving the optimal design solution is also improved by 80 percents. After the initial trial-run on solving identical problem, we apply our GA operators to various number classes and ratio of updated transactions. Results of these experiments are to be discussed in Chapter 5.

4.1. Enhanced Crossover Operator

Selecting proper crossover operator for specific problem type can reduce the number of generation, or converging rate, to reach optimal solution. Gorla chooses crossover operator on common edges, or edge-to-edge crossover operator, as the priority for solving this problem because it preserves adjacency information [15]. In this research, we proposed to adopt uniform crossover, which randomly choose one or more crossover bit(s) for crossing the selected parent. Uniform crossover preserves the relative order of the chosen elements in the first parent that imposed upon the second parent. Relative order of solution that preserves structure of

schema configurations is extremely important when the diversity of the individuals is small. Because such design solutions aims to satisfy frequent set of transaction's requests, the diversity of solution's access frequency is relatively small. Besides, feasible solution set bound design solution in small diversity of the individual, we are therefore interested to examine the uniform crossover, which preserves relative order of solution.

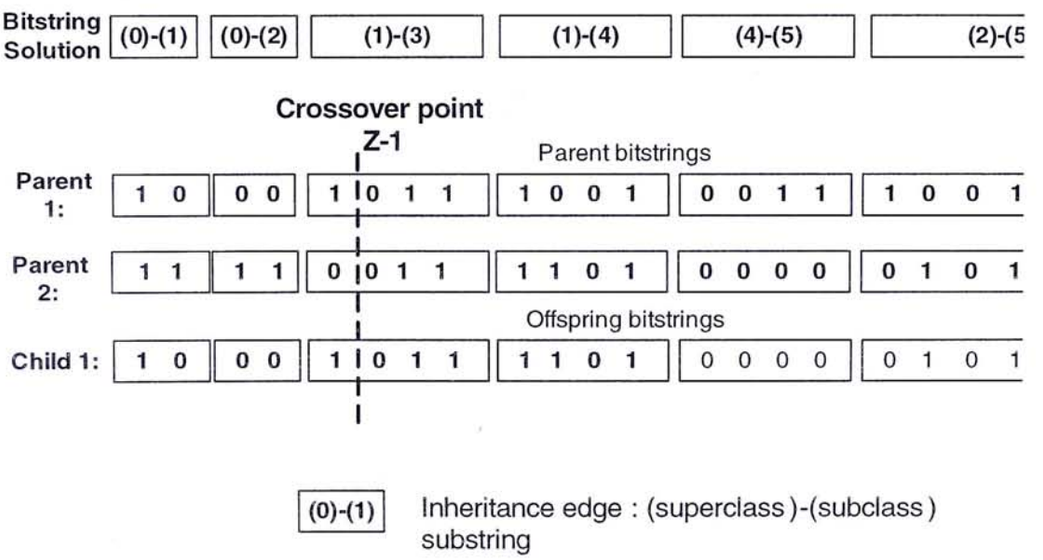


Figure 4.2: One- point crossover operation

One offspring solution is derived from parent solutions. Design solution is encoded to bitstring solution for performance evaluation. Enhanced Crossover Operation (ECO) is done by concatenates the substring segment of parent 1 with that from parent 2 to reproduce one child solution. Third edge is cut at Z-1 at the second bit of the third edge substring for 1-point crossing over. Child 1 is reproduced by concatenates parent 1's front segment '10 00 1' with parent 2's latter segment '011 1101 0000 010111'. Reproduced Child 1, which represent different solutions to the physical organization problem, are evaluated by objective function, which is database-operating cost for solution performance. Child 1 is checked for inheritance constraints. If violated solutions exist, Enhanced forward adjustment and backward adjustment are applied to the children solutions.

4.2. Infeasible Solutions and Enhanced Adjustment Method

In Gorla's implementation, bitstring generation, crossover position selection and mutation selection are all random processes. As these genetic operations may induce incorrect and incomplete representation to the design solution, we need an effective adjustment methods to repair the impair solutions. These solutions are adjusted in someway assure that they are feasible solution before the evaluation of fitness. Solution adjustment method is a problem-specific constraint handling method. Enhanced Adjustment Methods are designed to tackle with inheritance constraint solutions after enhanced crossover operation.

A number of individual solutions are randomly generated in the operation follows the evaluation of fitness on each solution. In this operation, an infeasible solution is the individual bitstring violated the constraint of application domain, which is inheritance constraint in physical design problem. In OODB design, inheritance constraint is applied in such domain. Individual solutions are encoded to 'genotypes' in bitstring form for genetic operations.

In traditional approach [32], inheritance constraint is applied to

each offspring solution checking for invalid solution. An infeasible bitstring solution contains one or more gene representation that violates the constraints of superclass-subclass inheritance. Before solutions in population are evaluated for fitness, a constraint handling method must be applied to "correct" or "evaluate" any infeasible solution so generated. Solution that violates inheritance constraints is called infeasible solution, which is handled in specific handling method. Enhanced Forward or Backward Adjustment Method for handling infeasible solutions in forward or backward direction after breeding operations is shown in Figure 4.3.

```
Input a bitstring solution
Create a new Class Hierarchy object that composed of edges;
If (input bitstring is not a valid solution)
        Input subclass inheritance edges;
        While (not reading last edge substring)
                Access to each edge for superclass and subclass tables;
                        While (not reading the last instance variable in
                        edge's superclass / subclass)
                        Count current bit position in the substring and
                        bitstring position;
                        If (current bit position inside forward / backward
                        segment && bitstring segment equals '0' / '1'
                        inherits subclass / superclass && superclass /
                        subclass is inherited upper edge/lower edge)
                                Set current bit position allele to '0' / '1'in
                                bitstring;
                                Set the upper / lower edge's corresponding
                                instance variable not / is inherits;
                                Count total number of adjustment made in
                                forward / backward;
                                Call function forward / backward
                                propagation result to trace all affected bits
                                across forward / backward substring edges;
                        End if
                End while
        End while
        return forward / backward adjustment result
End if
```

Figure 4.3: Forward and Backward Adjustment Method

Enhanced forward adjustment assumes the first segment is correct and modifications are made to the second segment so that inheritance constraint is satisfied. Backward adjustment assumes the second segment is correct and changes are made to the first segment so that inheritance constraints are satisfied. In the Figure 4.4 below, Child 1 is checked violated bitstring after the third common edge is crossing. Instance variables that represented by the second allele bit '1' in edge three are source of violation. We need to repair Child 1 before evaluation of fitness. Forward Adjustment method assumes instance variable in first edge is correct while instance variables being inherited in the third edge is subject to adjustment. The result of forward adjustment is shown in Forward Adjusted Child in which the fourth instance variable is adjusted to '0' that is highlighted. In similar manner, Backward Adjustment method assumes instance variable in lateral segment is correct while instance variable at second allele of first edge is subject to adjustment. The result of backward adjustment is shown in Backward Adjusted Child in which the second instance variable is adjusted to '1' that is highlighted.

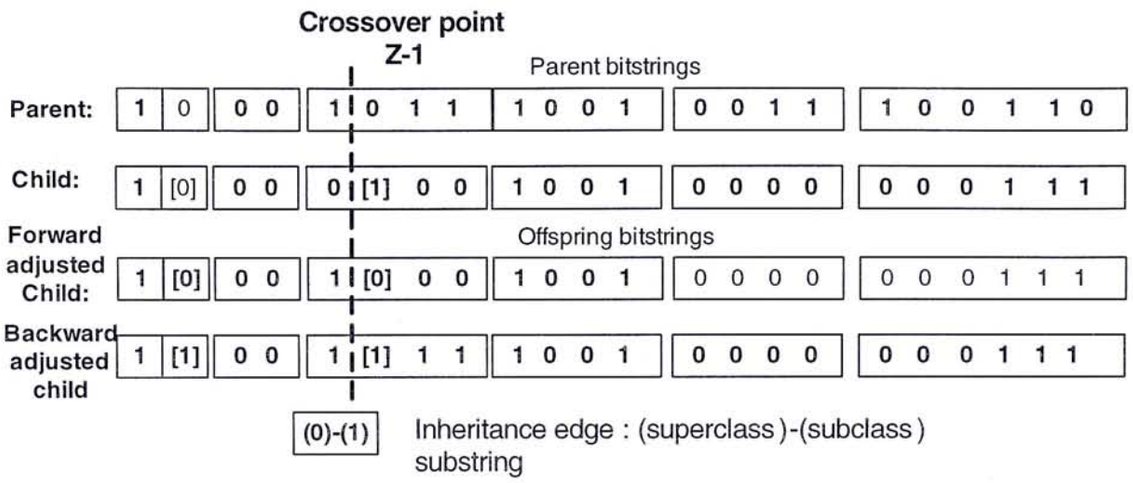


Figure 4.4: Forward and backward adjustment method

4.3. Propagation Adjustment Method

The operation of propagation adjustment start after the allele where crossover point is located. Enhanced Adjustment Methods walks on both directions to check for the number of inheritance violation. Checking starts from the crossover point. Upward direction check floats toward the root of main inheritance tree, while downward direction check sinks toward the leaf node of inheritance sub-tree. Results from these checking are the number of genes violated inheritance constraint. In the university database example, inheritance constraint is violated at second bit of the

sixth edge. This associated edge's second instance variable located at the fourth and the first edge is affected. After making the correction to second bit in edge four, Child 1' requires further adjustment to the first edge. Adjustment is propagation in backward direction until the adjustment reach the first edge. In similar manner, adjustment is propagation in forward direction until the adjustment reach the last edge. Results of these propagation adjustments in forward and backward directions are shown in the Figure 4.5.

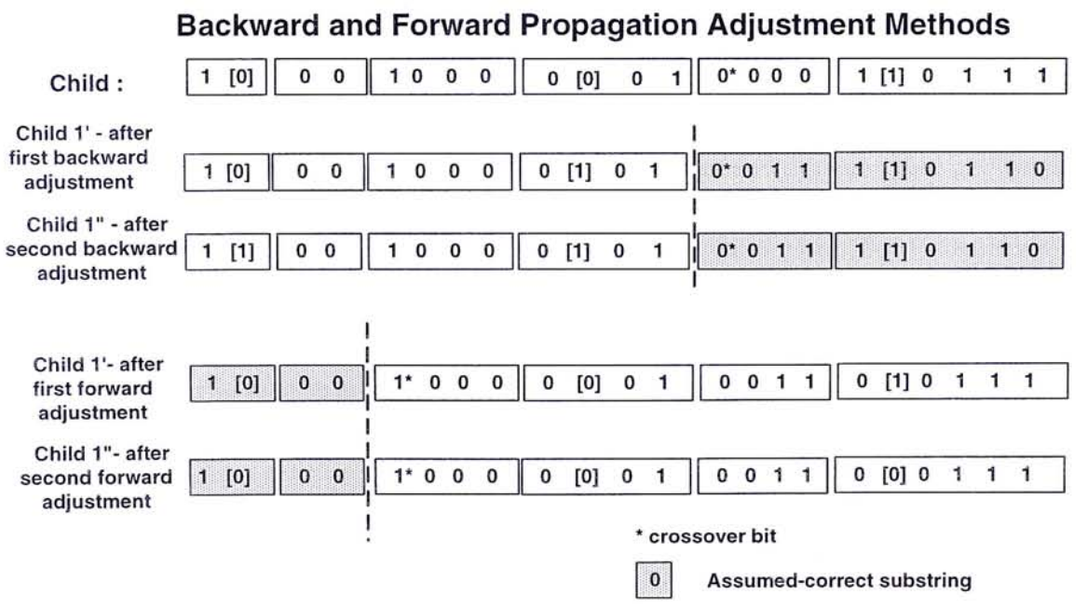


Figure 4.5: Backward and forward propagation adjustment method.

Chapter 5

5. Computational Experiments

5.1. Introduction

Properly designed physical storage structure can facilitate transaction's requests to access to classes in the secondary memory, which is costly in database operation. Physical organization problem for OODB has been solved with GA by N.Gorla [32]. Edge-to-edge crossover operator and two adjustment methods, forward and backward adjustments, are introduced provided. Edge-to-edge crossover operator preserves edge relation information during the generations but limited the searching set of feasible solution. This crossover operator limit to parent's edge crossing. It bounds the searching space to the edges of combination of in solution pool. It relies on mutation operators to exploit searching space, so that the performance of Gorla's solution with GA has low converging rate and lock on local optimal solution that is obviously ineffective search. Poor performance becomes prominent when the edge-

to-edge operator handles large number of classes in the hierarchy structure and therefore long computational time is resulted.

With the aim to exceed the confined search space of defined initial population and always maintain feasible solutions pool, one-point crossover operator and Enhanced Adjustment Methods (EAM) is introduced and applied for GA in this research. Experiment result reveals that these operator and methods accelerate the converging rate and reduce computational time in solving for feasible solutions. Forward adjustment and Backward adjustment check inheritance constraints generated in populations while new generation is converging to universal optimal solution. One-point crossover uses unbiased selection of instance variable from the edge as the crossover point, where cutting edge's crossover composition increases the rate of solution convergence. It is also important to maintain solution feasibility while the solution pool is converging to the optimal solution. EAM ensures solution feasibility during processing each generation, which checks inheritance constraint and corrects infeasible solutions so as to guarantee feasible solution pool.

In this research, we propose Enhanced Adjustment Methods that improves traditional method for large class hierarchy handling capability and improving the rate of converging to optimal design solution. EAM traces inducing inheritance constraints after forward and backward adjustment in the solution children pool.

5.2. Experiment Objective

In this research, we propose Enhanced Adjustment Methods that improves traditional method for large class hierarchy handling capability and improving the rate of converging to optimal design solution. We provided an Enhanced Crossover Operator and Propagated Inheritance Constraint Method for tracing the induced inheritance constraints after forward and backward adjustment in the solution children pool. On the comparison basis test, crossover solution and mutation solution adjusted by Forward and Backward Adjustment Algorithm compare to that of death penalty algorithm. Higher rate of converging to optimal design solution meet the OODB nowadays requirement of frequently updated transaction pattern. In the same Gorla's university database with 6 classes as an example, the optimal design solution is 32 in the ninth iteration. Gorla's crossover method reach 35 in the third iteration and 34 in the sixth. Proposed ECO reach 34 in the fifth iteration and reach optimal solution 32 in the seventh. We can see the Enhanced Crossover Operator (ECO) and Enhanced Adjustment Method (EAM) has improved the rate of solution convergence while guarantee the solution feasibility in the population. Besides, Enhanced Crossover Operator and EAM can handle database with over 25 classes while maintaining the feasibility of design solutions. This is a significant improvement for OODB which promise the handling of large and complex class hierarchy.

5.3. Tools and Setup

We use GA Playground [17] as a toolkit to conduct our experiments. It is a general purpose genetic algorithm toolkit implemented in Java language. We define and run our own optimization problems. We have also extended or re-written some classes (i.e. GaaFunction, GaaCrossover, GaaMutation) to create a different genetic algorithm with different fitness function.

The implementation of the genetic algorithm uses a high alphabet to encode the chromosome's genes. In this implementation, each locus on the chromosome stands for a complete gene or variable.

The GUI of the GA playground is shown as in Figure 5.2. It provides a graphic window which can optionally be used for displaying graphic representation of the evolutionary process. Parameters (i.e. Population size, number of genes) can be set manually to suit for different experimental setups, see Figure 5.3 for illustration.

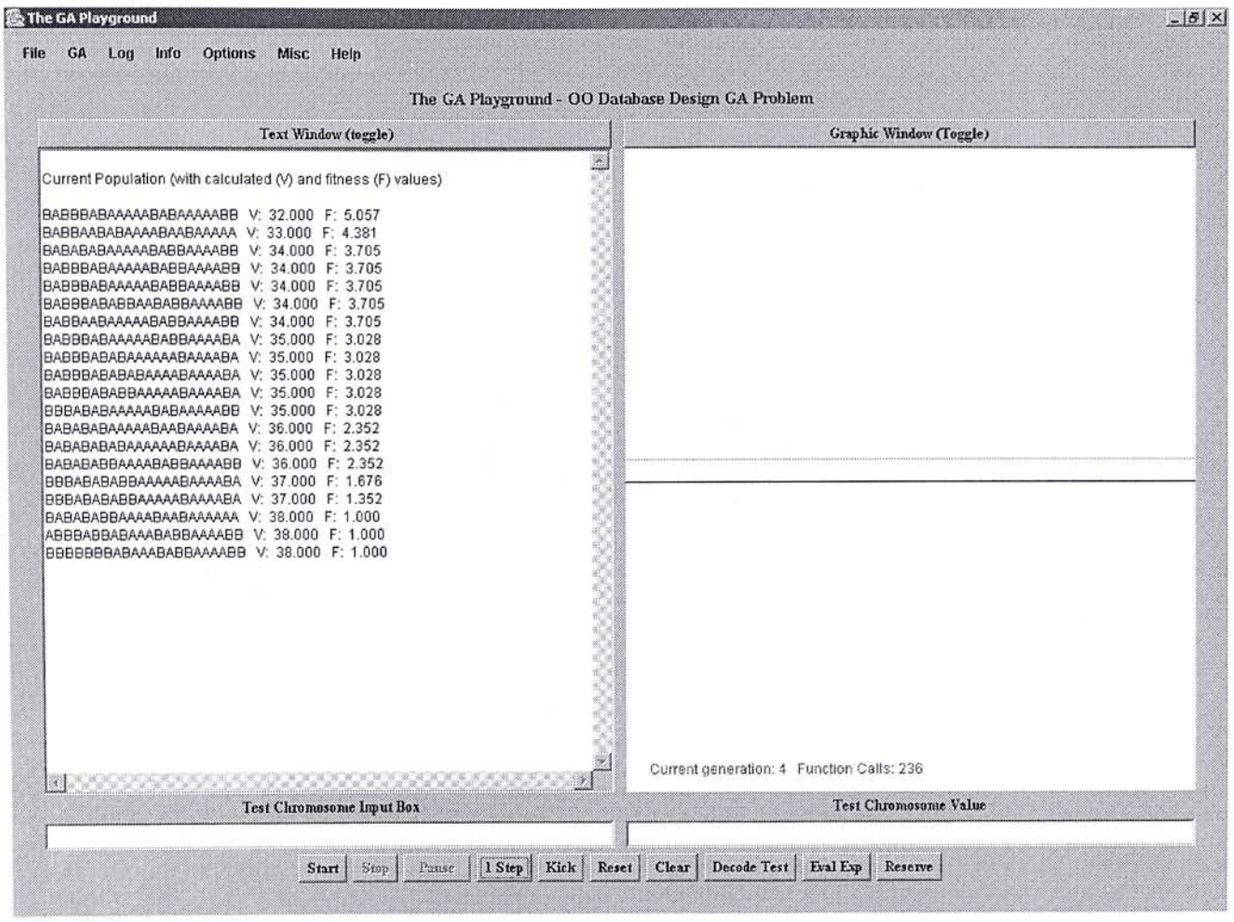


Figure 5.2: GUI of the GA Playground

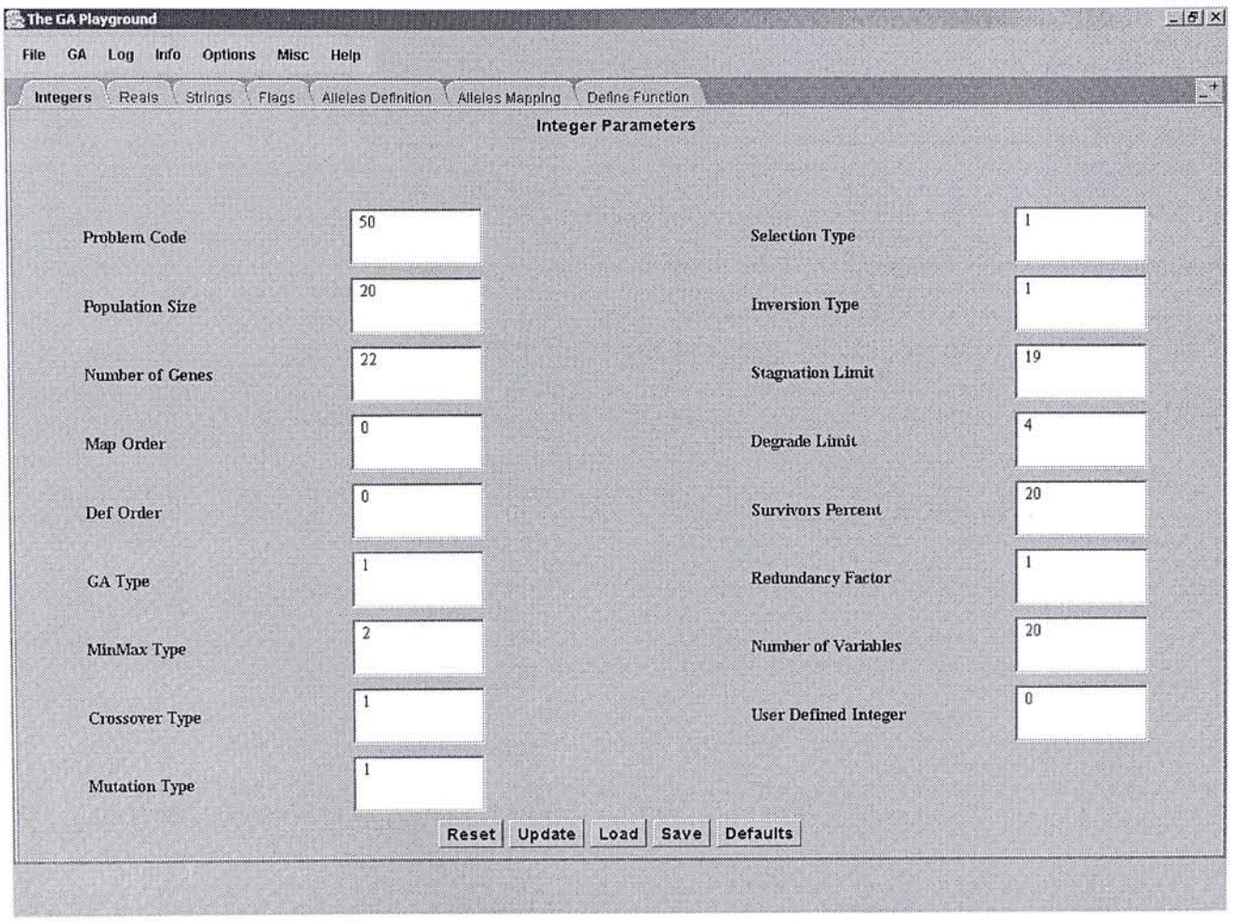


Figure 5.3: Interface for parameter settings

5.4. Crossover Operator

We applied GA Playground's crossover operator, which adopted crossover strategy with less parent selection force but high converging force. Eshelman's CHC [15] uses unbiased reproductive selection by randomly pairing all the members of the parent population to creates one child, and then replacing the worst individuals of the parent population with the better individuals of the child population. In this research, we proposed to apply Eshelman's CHC approach, so that the selection force is maintained even if the converging force is weak. It is obvious that the selection force is weak when class size is large, because range of parent's fitness is not prominent even scale to a specific fitness range. We adopt to higher converging force.

5.5. Mutation Operator

Mutation is done by selecting one bit at random and deciding to flip the bit or not, by using a random number. Perform forward and backward adjustments, if necessary, to satisfy the inheritance constraint. Mutation is done to one bit out of 400 bits, resulting a mutation probability.

5.6. Termination condition

Compute database operating cost, fitness of solution, and selection fact for each of solutions. Generation repeated until the crossed-over solutions are worse than the parent solutions; algorithm may be stopped after a certain time is elapsed and the best solution from the pool is considered the desired solution.

5.7. Computational Experiments

We implemented our Enhanced Crossover Operator (ECO) and Enhanced Adjustment Method (EAM) using the GA Playground [17]. The algorithm is implemented using Java and the program ran on a PC with 1.0 GHz processor. Several experiments are conducted to illustrate the effectiveness and efficiency of our approach towards database storage design.

5.7.1. An Illustrative Example --- UNIVERSITY database

We consider the OODB logical schema of University database that shown in Figure 3.4 in Chapter 3. Table 5.1 lists all the retrieval and update transactions that are considered on the database. Frequency of these transactions is assumed to be 1 each [32].

Transaction	Type of transaction
1. Get all details of teaching-assistant with SSN = 498	RETRIEVAL
2. List all faculty names, ranks, and their salaries	RETRIEVAL
3. List all the names of students and their GPA	RETRIEVAL
4. Increase the salary of all employees by 5%	UPDATE
5. Modify the GPA to 3.5 for student with SSN = 450	UPDATE
6. Increase the age of staff#234 by 1 year	UPDATE

Table 5.1: Details of the retrieval and update transactions

A typical solution consists of a bitstring of 1's (denotes the corresponding instance variable is to be stored in the subclass) and 0's (denotes otherwise). The solution bitstring in this example contains six substrings that relate to the six superclass-subclass edges, as depicted in Table 5.2.

The edge on the graph	Possible variable inheritance	Example substring
1. PERSON → EMPLOYEE	Name, Age	10
2. PERSON → STUDENT	Name, Age	00
3. EMPLOYEE → STAFF	Name, Age, Date-Hired, Salary	1011
4. EMPLOYEE → FACULTY	Name, Age, Date-Hired, Salary	1001
5. STUDENT → TA	Name, Age, Major, GPA	0011
6. FACULTY → TA	Name, Age, Date-Hired, Salary, Rank, Dept	100110

Table 5.2: The edges and their corresponding example substring

The initial solution pool for the University database example is summarized in Table 5.3. The average cost of solutions in the solution pool is 44, while the best cost is 34.

Solution #	Solution bitstring	Cost
1	10110011000000000000011	34
2	10111001001000100000011	35
3	00100010001010100001011	37
4	10111010101100111010001	38
5	10001010100000101000001	39
6	10000011001000000000001	41
7	10001001101000110010001	42
8	01110010000110000000011	43
9	01100011011010010100011	43
10	10100001000100100000010	44
11	00000000001000110000010	44
12	10101011000110000000011	44
13	11111110011100010110000	46
14	11011101010001010000001	47
15	00110011001111111001110	47
16	10000011001100100000001	48
17	11111001001111000001100	48
18	00000001000100000000001	49
19	11001010000100110001000	54
20	01000101001100110011101	57
Total		880

Table 5.3: Initial solution pool

By randomly pairing all the members [15], ten pairs of parents are selected for mating from the initial solution pool. For each pair of parents, one bit is selected at random as a crossover point. By crossing over each pair of parents at the crossover point, one offspring is produced for each mating process.

To avoid premature convergence and slow finishing, Survival-rate parameter

define certain percentage of better-fit parent genes survives to the next generation. Since the survival rate is 20%, only 4 out of 20 bitstrings will survive to the second generation solution pool. The 4 bitstrings are grayed in Table 5.3 and 5.5.

One of the mating process is shown in Table 5.4. For example, solution #2 and #17 are selected for mating from the initial solution pool. Crossing over these parents at crossover point 14 produces an offspring which is found to be infeasible. The offspring is then perform forward and backward adjustments at the randomly selected point 16. The forward adjusted offspring and backward adjusted offspring will undergo mutation, which is done to 1 bit in each bitstring. Since the resulting mutated offsprings are feasible, no further adjustment is needed. Both the mutated offsprings are "good" enough to enter the second generation solution pool.

Solution #2 *10111001001000*10000011 Solution #17 111110010011*100000110*
Crossover (crossover point: 14) Offspring: 1**0**1110010**01**00010000**1**10
Adjustment (adjustment point: 16) After forward adjustment: 101**1**10010010001000**0**10 After backward adjustment: 1111100**111**10010000110
Mutation Mutated forward adjusted offspring: 101**0**10010010001000010 Cost = 36 Solution #8 in second generation solution pool Mutated backward adjusted offspring: 111110**1**1111100100001 10 Cost = 44 Solution #20 in second generation solution pool
No further adjustment is needed since they are feasible

Table 5.4: Example of mating process

As we can see in the Figure 5.4, the average cost of solutions in the solution pool is 38.1, while the best cost is 34. This represents improvement in the solution pool.

The process is repeated and the optimal solution is obtained in the fourth generation. The best solution has a cost of 32. The generation wise average and the best costs are shown in Figure 5.4. It is found that our result is better than Gorla's one, which requires ninth generation to reach optimal solution. Our GA algorithm

can makes the minimal cost of solution converge at faster rate in generations. Our GA approach provides an efficient storage structure that is very close to optimal solution and is a substantial improvement compared to the Gorla's approach.

Solution #	Solution bitstring	Cost
1	11111010011010100011011	34
2	10110011000000000000011	34
3	11111010000001000000011	34
4	10111001001000100000011	35
5	10110000001010110000010	35
6	10110000001010110000010	35
7	10110011000000000000001	35
8	10101001001000100000010	36
9	00110010001010100011011	36
10	00100010001010100011011	37
11	10111010101100111010011	38
12	10000010001000000000001	39
13	11111111011110000100001	40
14	11000010001000100011011	40
15	10000000001000110000010	40
16	10001001101000000000010	41
17	00000010001000100011011	42
18	01110010000011000000001	43
19	11111001111101000001100	44
20	11111011111100100001100	44
Total		762

Table 5.5: Second generation solution pool

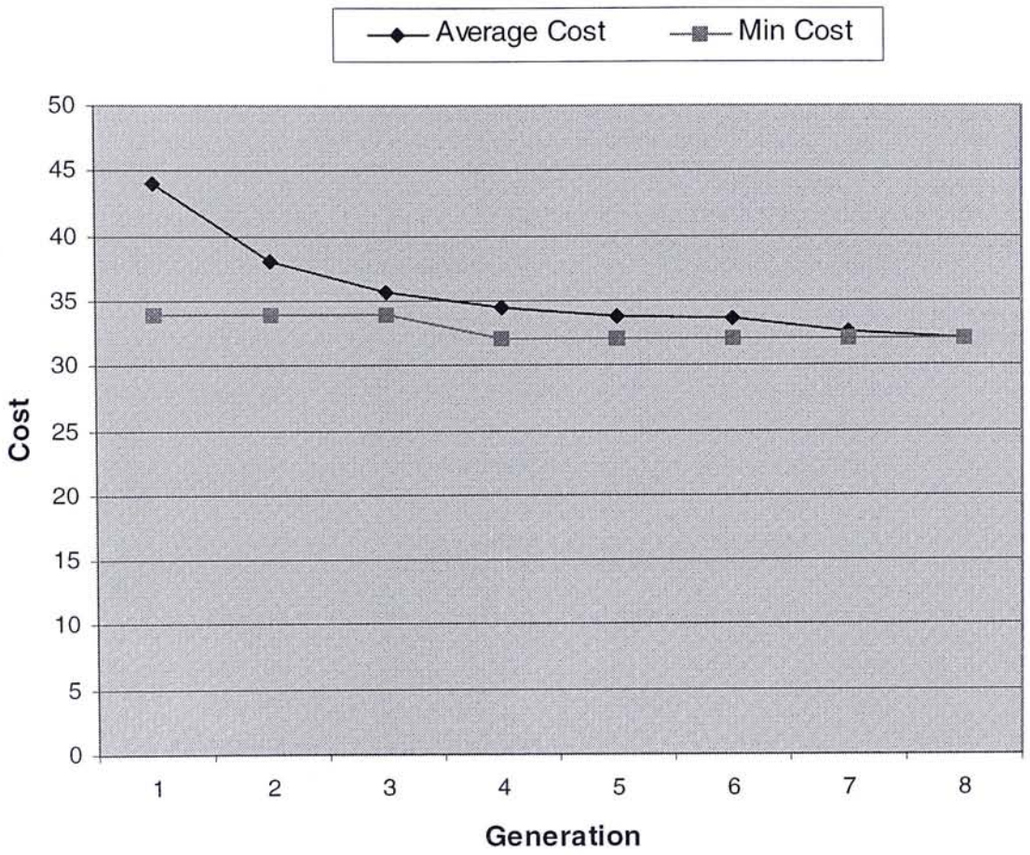


Figure 5.4: Generation wise database operating cost of UNIVERSITY database example.

The bitstring of optimal solution is "10 11 1000 0010 1010 001011". The storage structure corresponding to the best solution is shown in Figure 5.5. The grayed variables are those inherited from direct or indirect superclasses.

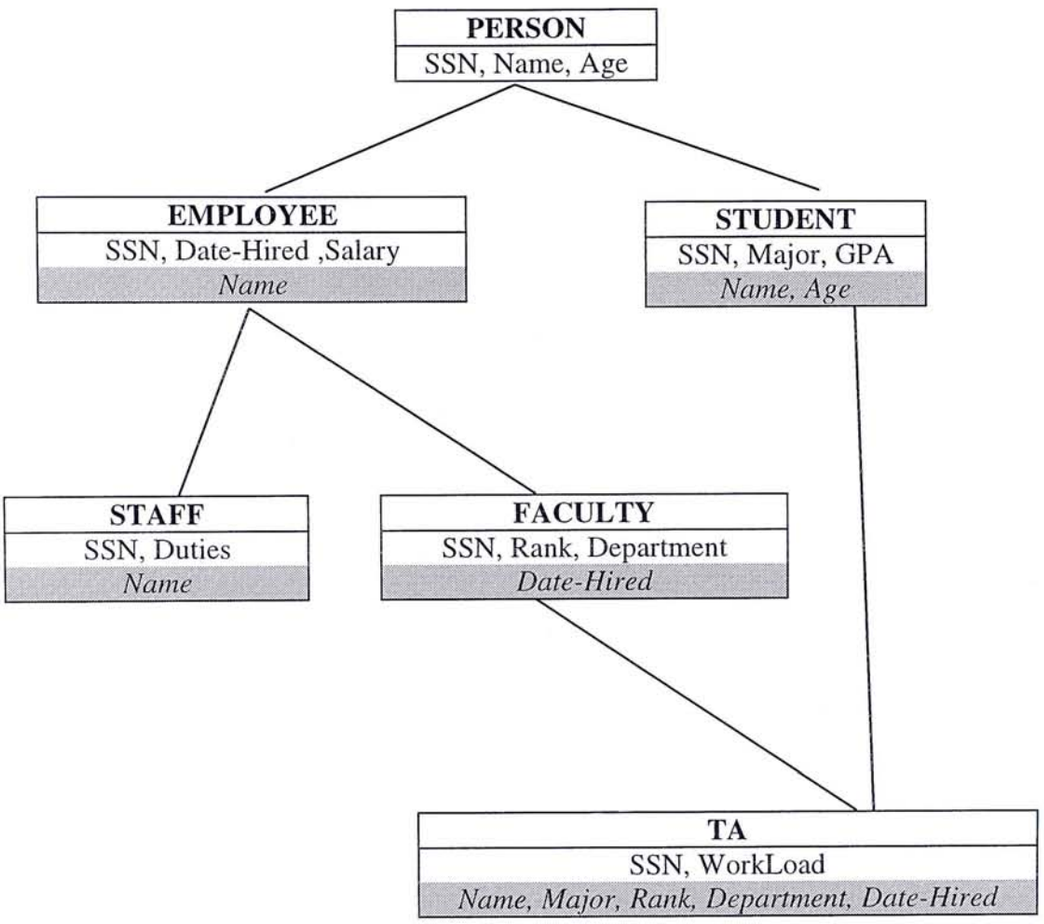


Figure 5.5: An optimal OODB storage structure for University database.

5.7.2. Simulation --- 9 classes and 25 classes

In order to prove our approach can handle large number of classes, we use GA Playground [17] to automate the database design process using GA. The parameters are listed in Figure 5.6. The number of instances in a subclass is determined using random number between 50% and 100% of the number of instances in its superclass that has the least number of instances. Each transaction is generated considering logical OODB model, such as in Figure 3.4. The target class for a transaction is chosen randomly from the classes in the database schema. Then all the superclasses of that target class are identified. Some of these superclasses are selected at random, instance variables are then selected randomly from these selected superclasses. The number of instances needed for a transaction is determined from the number of instances available in the target classes. This constitutes a transaction.

Our GA algorithm is applied on schemas with up to 25 classes. The proportion of updates is 50% with 3 UPDATE and 3 RETRIEVAL transactions. As described in Chapter 3, we use the number of instances needed access is used as a measure of database operating cost.

Number of classes in the C-H<=25

Number of home instance variables in a class<=15

Number of instances in a class<=1000

Number of solutions in the GA pool=20

Number of transactions=6

Figure 5.6: Parameters file

5.7.3. Result

The generation wise average and the best costs for 9 classes and 25 classes databases are shown in Figure 5.7 and 5.8 respectively. The optimal solutions for 9 classes and 25 classes have costs 6580 and 8073 respectively. The optimum for 25 classes is first obtained in the eleventh generation. As compared with Gorla's fourth generation, ours converge in a faster rate.

The average time taken for GA to solve a database problem with five classes and nine classes is 1.1 min and 8.2 min respectively, while it is 40.7 min for the schema with 25 classes. As compared with Gorla's 1.4 min for five classes and 60 min for 25 classes, ours processing time is faster.

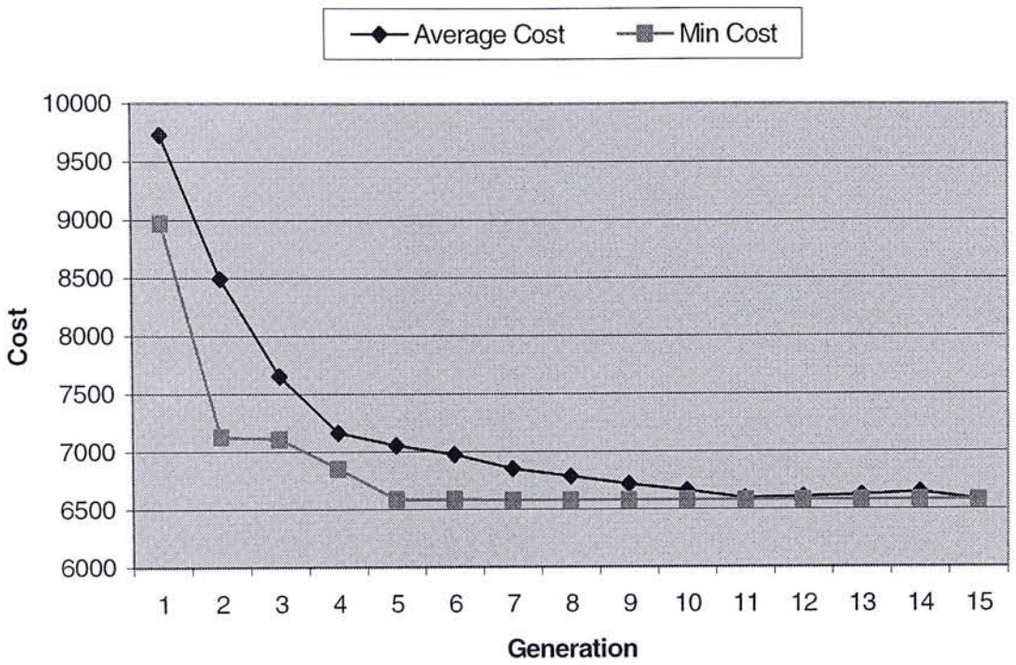


Figure 5.7: Generation wise database operating cost for 9 classes.

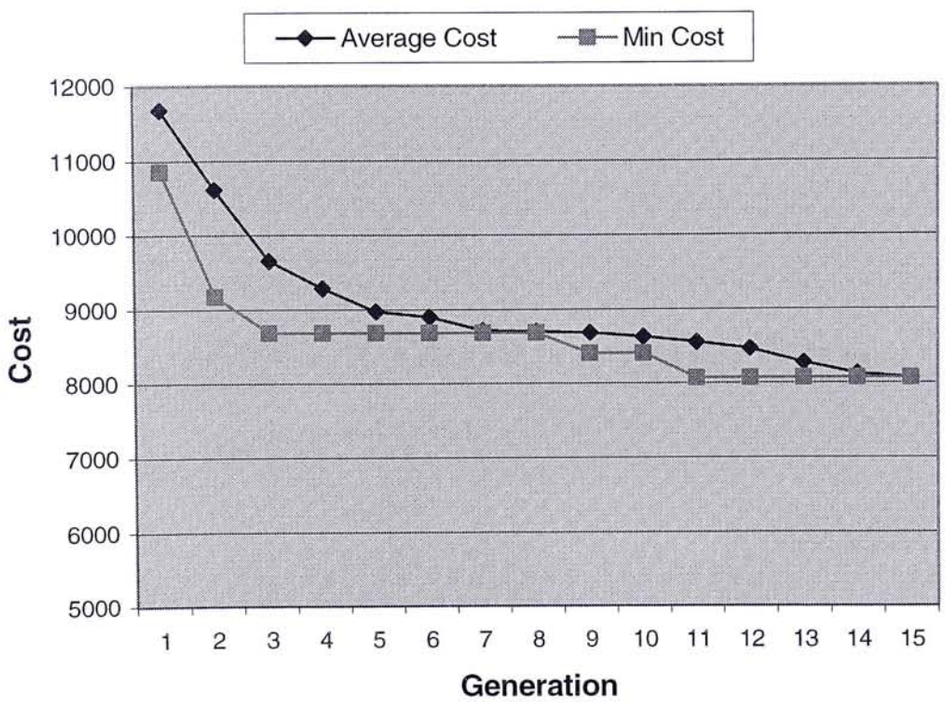


Figure 5.8: Generation wise database operating cost for 25 classes.

Chapter 6

6. Conclusions

6.1. Summary of Achievements

Object-oriented databases are known to be rich in functionality but poor in performance, especially in processing user's transaction requests for update and retrieval of information. Poor organization of physical storage structure causes transactions frequently access to classes for object instance data in the secondary memory. This process incurs high database operating cost and reduces the performance of OODBs. A methodology was devised for solving this physical organization design problem with GA by N.Gorla. Although GA has solved this physical storage design problem, the GA performance is poor in terms of converging rate and computational time.

Solving this physical database design at higher converging rate can meet the OODB nowadays requirement of frequently updated transaction pattern. In our approach, we adopt the objective function from Gorla to test various GA operators' performances in solving for this design problem. In this research, after considering problem-specific characteristics, we propose to adopt different GA methods, including generation of initial population and breeding, parent selection, and replacement selection. Gorla's solution with GA is poor in terms of converging rate and computational time. This research revealed that GA performance could be improved after using the proposed Enhanced Crossover Operator and customized GA operators. Enhanced Crossover Operator and Propagation Adjustment methods are proposed for better GA performance.

Experiment result reveals that these operator and methods improve the converging rate and reduce computational time in solving for feasible solutions. In the same Gorla's university database with 6 classes as an example, the optimal design solution is 32 in the ninth iteration. Gorla's crossover method reaches 35 in the third iteration and 34 in the sixth. Our design solution is close to optimal solution (within 5%) after three generations for solving the same university database problem adopted from Gorla. We obtained optimal solution after four generations of the GA. The optimum for 25 classes is first obtained in the eleventh generations. Comparing that to Gorla's GA approach the optimum at the fourteenth generation, ours GA approach has improved the converging rate.

GA Playground is an open-source GA package written in Java-program for researchers to test GA in solving complex optimization problem. Objective function for the evaluation of design solution, propagation adjustment method for infeasible solutions, and several modifications over GA operators were made to the GA playground. For time performance, the time taken for GA to solve a database problem with 9 classes is 12.2 seconds, while it is 50.8 seconds for the schema with 25 classes. This is a significant improvement for OODB performance which promise the handling of large and complex class hierarchy.

7. Bibliography

1. C. C. Low, B. C. Ooi, H. Lu, "H-trees: A Dynamic Associative Search Index for OODB", *Journal of ACM SIGMOD*, pp.134-143, 1992.

2. C. H. Cheng, Y. P. Gupta, W. H. Lee and K. F. Wong, "A TSP-based Heuristic for Forming Machine Groups and Part Families", *Int. J. Prod. Res.,* vol. 36, no. 5, pp. 1325-1337.

3. David Beasley, David R. Bull and Ralph R. Martin, "An Overview of Genetic Algorithms: Part I, Fundamentals", *University Computing*, 1993, 15(2) 58-69.

4. D. B. Fogel, L. J. Fogel, and J. W. Atmar., "Meta-evolutionary Programming", in R. R. Chen, editor, *Proc. 25th Asilomar Conf. on Signals, Systems and Computers,* pp. 540-545, San Jose, CA, Maple Press, 1991.

5. D.D. Straube, M.T. Ozsu, "Query optimization and execution plan generation in object-oriented data management systems", *IEEE Transactions on Knowledge and Data Engineering,* 7 (2), 1995.

6. D. E. Goldberg, "Computer-aided Gas Pipeline Operation Using Genetic Algorithms and Rule Learning", *(Doctoral dissertation, University of Michigan), Dissertation Abstract International*, 44(10), 3174B, (University Microfilms No. 8402282), 1983.

7. D. E. Goldberg, and R. E. Smit, "Blind Inferential Search with Genetic Algorithms", paper presented at *ORSA/TIMS Joint National Meeting*, Miami, FL, 1986.

8. D. E. Goldberg, "Computer-aided Gas Pipeline Operation Using Genetic Algorithms and Rule Learning", *(Doctoral dissertation, University of Michigan), Dissertation Abstract International*, 44(10), 3174B, (University Microfilms No. 8402282), 1983.

9. D. E. Goldberg, "Genetic Algorithms in Search , Optimization, and Machine Learning", (Reading, MA: Addison-Wesley)

10. D. E. Goldberg, and R. E. Smit, "Blind Inferential Search with Genetic Algorithms", paper presented at *ORSA/TIMS Joint National Meeting*, Miami, FL, 1986.

11. D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W.Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, M.C.Shan, "Iris: an object-oriented database management system", *ACM Transactions on Office Information Systems* 5 (1), 1987.

12. D. Maier, et al., "Development of an Object-Oriented DBMS", *Proc. 1st OOPSLA Conf.,* pp. 472-482, Portland, OR., 1986.

13. D. W. Cornell and P. S. Yu, "A Vertical Partitioning Algorithm for Relational Databases", *Proc. 3rd International Conf. On Data Engineering,* pp. 30-35, 1987.

14. D. Whitley, "Using Reproductive Evaluation to Improve Genetic Search and Heuristic Discovery", in J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pp 108-115, Lawrence Erlbaum Associates, 1987.

15. Eshelaman L J, "The CHC adaptive search algorithm: how to have safe search when engating in nontraditional genetic recombination Foundations of Genetic Algorithms ed G J E Rawlins", *San Mateo, CA: Morgan Kaufmann*, pp 265-83, 1991

16. Gajanan S. Chinchwadkar, Angela Goh and Ee-Peng Lim, "Simulated Annealing for Vertically Partitioning an OO Database", *International Conference on Information, Communications and Signal Processing,* ICICS'97 Singapore, 9-12 September 1997.

17. http://arieldolan.com/ofiles/ga/gaa/gaa.html#Application (GA Playground Reference)

18. H. Ishikawa, "An Object-Oriented Knowledge Base Approach to a Next Generation of Hypermedia System", *Proc. IEEE COMPCON Conference,* pp. 520-527, 1990.

19. H. Ishikawa et al,, "A Knowledge-Based approach to Design of Portable Natural Language Interface to Database Systems", *Proc. 35th IEEE Data Engineering Conf.,* pp. 134-143, 1986.

20. H. Ishikawa et al., "KID: Designing a Knowledge-Based Natural Language Interface", *IEEE Expert,* vol.2, no.2, pp. 57-71, Summer 1987.

21. Hong K. Chung and John P. Norback, "A Clustering and Insertion Heuristic Applied to a Large Routing Problem in Food Distribution", *J. Op. Res. Soc.,* vol. 42, no. 7, pp. 555-564, 1991.

22. Holland J H, "Adaptation in Natural and Artificial Systems", *Ann Arbor, MI: University of Michigan Press*, 1975

23. J. Banerjee, H.-T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballous, "Data model issues for object-oriented applications", *ACM Transactions on Office Information Systems* 5 (1) (1987).

24. M.J. Willshire, H.-J. Kim, "Properties of physical storage models for object-oriented databases", *IEEE conference on Databases, Parallel Architectures, and Application* , 1990.

25. M.J. Willshire, "How spacy can they get? Space overhead for storage and indexing with object-oriented databases", *Proc. 7th International Conf. on Data Enginnering, pp. 14-22,* April 1991.

26. M.S. Oliver, S. Von Solms, "A Taxanomy for Secure Object-oriented Databases", *ACM Transactions on Information Systems* 8 (4) (1994).

27. O. Duex, et al., "The Story of O2", *IEEE Trans. Knowledge and Data Engineering,* vol. 2, no. 1, pp.91-108, March 1990.

28. J. J. Grefenstette, "Optimization of Control Parameters for Genetic Algorithms", IEEE Trans. SMC, 16, pp. 122-128, 1986.

29. J. N. Bhuyan, V. V. Raghavan, V. K. Elayavalli, "Genetic Algorithm for Clustering with an Ordered Representation", *Proc. 4th International Conf. on Genetic Algorithms,* in R. K. Belew, L. B. Booker, editor, Morgan Kaufmann, 1991.

30. L. Davis, "Handbook of Genetic", *Van Nostrand Reinhold*, 1991.

31. N. Ballou et al., "Coupling expert system shell with an object-oriented database system", *Journal of the Object-Oriented Programming*, pp. 27-44, 1992.

32. Narasimhaiah Gorla, "An object-oriented database design for improved performance", *Journal of Data and Knowledge Engineering*, pp.117-138, 2001.

33. Kirkpatrick. C, Gelatt C., Vecchi M., "Optimization by Simulated Annealing", *Science 220,* 1983.

34. R. Gupta, E. Horowitz (Eds.), "Object-Oriented Databases with Applications to Case", *Networks, and VLSI CAD, Prentice-Hall, Englewood Cliffs, NJ*, 1991.

35. R. L. Haupt, S. E. Haupt, "Practical Genetic Algorithms", *John Wiley & Sons*, 1998.

36. S. Khoshafian, "Insight into object-oriented database", *Information and Software Technology 32*, pp 274-289, 1990.

37. S. Navathe, S. Ceri, G. Weiderhold, J. Dou, "Vertical Partitioning Algorithms for Database Design", *ACM Transactions on Database Systems 9 (4)*, 1994.

38. Syswerda G, "Uniform crossover in genetic algorithms", *Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, 1989,) ed JD Schaffer (San Mateo, CA: Morgan Kaufmann)* pp 2-9, 1989

39. T.Back, D.B.Fogel, Z.Michalewicz, "Evolutionary Computation I", *Institution of Physics Publishing*, 2000

40. W. Kim, et al., "Architecture of the ORION Next-Generation Database System", *IEEE Trans. Knowledge and Data Engineering,* vol. 2, no. 1, pp.109-124, March 1990.

41. W.W. Chu, I.T. Ieong, "A Transaction-based Approach to Vertical Partitioning for relational database systems", *IEEE Transactions on Software Engineering 19 (8),* 1993.

42. W. Kim, "Object-oriented databases: definition and research directions", *IEEE Transactions on Knowledge and Data Engineering*, 1990.

43. W.K. Lee, C.H. Cheng, "Genetic Based Clustering Algorithms and Applications", *Master of Philosophy dissertation, Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong,* 2000.

44. Z. Michalewicz., "Genetic Algorithms+Data Structure = Evolution Programs (Third Edition)", *Springer Publishing*, 1999.

45. J.E. Baker, "Reducing Bias and Inefficiency in the Selection Algorithm", in *J. J. Grefenstette, editor, Proceedings of the Second international Conference on Genetic Algorithms*, pp. 14-21, Lawrence Erlbaum Associates, 1987.

46. B. Li, W. Jiang, "A Novel Stochastic Optimization Algorithm associative search index for OODB", *IEEE Trans. on Systems, Man, and Cybernetics, Part B, 30(1),* 2000.

47. K. Y. Tam, "Genetic Algorithms, Function Optimization, and Facility Layout Design", *European Journal of Operation Research 63(2)*, 1992.

8. Appendix

A. Initial population for 9 classes

Solution#	Solution bitstrings	Cost
1	100101011000000110000101100001101111100010100001000 101000000100000000010001000000011011000000100000000 011111001100100000000000011000010000010000001100000 000001001000001101001100100010100000000100101000100 010000100110001000000001000000000111110001000100100 000000010001001101111000000000100000010000000001000 000010010100010101110100000101001010000000010000	8972
2	101011110010101110000011110110000111000001110010011 101000000001100000000011100000010111000000000100001 111001010010010100000000010000001000010100000001000 000000000010001011100010000001100000001000001100100 011001011100000010000101000110001101000010100101010 000000000111010001100000000000001000010001000000000 000010001001001110110110001001010000000000010011	8976
3	010010101100000101001010101001001111110100000100000 000100011001010000000000000000010001001100101000101 1 000000000011001000000000100111000001010010000000000 100000001000101001000100010000000000010010100010000 010001000000100000000000000110000000000000000011000 000000000000001001101010001001000000010010000100010 000000000000000010101000011000000000000000000001	9230
4	011100100101011000000100010100001010100010001011101 100100110101001000100011001000000000100100000000110 001100110110000000000100010001000001001001000000000 000010001110001000001010100000001000000000100010100	9234

	000000000001000000100010010000010100110000100100000 000000011000000001100100010100000000010000000100010 100000001000000001000010100000101000110000000100	
5	101000011000001010100111001110100110100000000100011 110000100000110000000110000001101000000000100000011 100110010111000110000000011101000000000111010100000 000100001010000100101000000011100100000010000000100 011011010000000000000010101011100010011000010010000 100000000111110101101100100000010000010000010000000 000001010010000001010110000000001000100000010100	9234
6	010001001001010100010000101001000101000000010100010 000000101000000000000100001001101000100000000000011 100010000100000000000000000100010101000100110100010 000100000110001101010010000000000100001010001010000 000011010000000001000011000010001000000100000010000000 000000000100001001101010101000000000010000000001000 000000011010001000000110000000100001000000000000	9488
7	110111010110010101101110111000000011001000001001110 100001001000100100101100000000100100000010000000111 110110000001110000100001000000000000011100010010000 001000000000100101001111101000011000001010001000000 101000000000000100000000000001111111000000000001000 000100011101000001100100000010100100010001000000000 000001000100000100000100001000100000000000000000	9760
8	010111000010001000000101000010001000000010100000110 100001011000000100000100000000000000000000001000001 110110100110000000000000010001010000000110000000000 000100000000100000001110000010100000001000001010001 010000000000000000000000100111101100000010001000000 000000001101000001110000011000001000010000000000010 001010000000010100010110100000100000000010000101	9764
9	010110001100000110010111111001111000100000110101010 010000000100010000010000101001010000000000001100011 100010100101110011000000010100101101010110100000000 010000001100001100010110100001000000001000010000101 010110000000000000101100100101000100001010010101000 011000010100101001110001000000011100010000000000000 101000101110000101010110010010000000000100000001	9778

10	10010010000011100111110010010001111000010001001001 0 101000101101000001100100010000011100000101000000011 1111010000011010000011001000000010001010011000100000 0001000010000011000000111010100010001001010001010001 000100111000000000000100000000111000110010000010101 000001100101010001110000001101000000110001010000000 0010001011000111000000100100010000000100010000000	9778
11	10001000111111110011111011001110001000100000001000 01000000100001010000000001000010000000001000010000011 11001100000100000111000000000100001000000011100000000 100000100010000101101011000111001000001010101000101 011010000000010000010101010111010100001100000001000 0011100100001010111000100000000100000000001000110000 000001001110010100000010001000000100100000010101	9778
12	101101100101001000011111110110001100100111011001111 1000101001000000100010110100000110001001010000000001 001100011111100001010010000100010000010100001100000 001011010000000100001110000001100011011000000010101 001000100001000101101001010001010101100100010110100 001010011111010001111000000101010010000001000000001 0010000000000000100010111000000000100001100010100	9778
13	001101110101101100111101000110000011000000100010101 0000001001001010010000000000000000100000100100000000 1100000100000001010100000001100010001000010000000000 00000000001000010001000000000100000001000110001000000 0010000000000000000000000100101000001001000000000000 0000000010100100011001101001001000010100000000001000 00000001100000000100011000001000000000100000100	9778
14	000100101000110000101010000100001101001100000001101 000000110000010000111000000001110000000010010000011 1101000000000100010001001000110010101000000011000000 0110000010100011000010111010000000010100000001010101 011010100000001000000000101001100010000001000000001100 010001011010000001110000110000000000000000000001000000 0000100000000100010000100011000000000000100010101	9778
15	101111101110011001010010111110001011100000000101001 100011010101011100000000010010000010010100000101001 011110001101100000100100000001010000000110000001000 00000000010111110010011110000001110001010101100101 0000000000000000000000001010101000111101100001001000 0001000100110100010001000100000000000000001010001100 00000001110001010100001000000000100010100000110	10036

16	0111001010000111000010100100110010011000000001001011 0101100010010000001100000000000010000001001000000000 1100001001101000100001001001010101000100000000000000 1000000010000000110001011000000000000100001000000001 000000000000000000000101000110011110000101000101100 0100001101101111011100010010010000000100100000000000 0010000010000000101000100000000000100000100000010	10036
17	1010010101000011110110011100011110011000000001001001 1000010000000100000001000000001000000001000000100010000 0010100110100000000000000110010100000000100000000000 0000000000010010000011001001001100000001110001001000 0110000000000000000000100100100000000110010000001000 0000000100000000110100011000000100010010000100000001 00001000000001000100000000000000000010101000010	10308
18	1000010001110101010101011101010111101100101100111100 1100010000000010101000011000101001000000100000000011 1101100000001000000110001000000010001010100010000000000 000000000100010010111000000101010110010101010100 0111010010000000001100010101001100100110000000011000 001000011001100011111000000100001000010001000101010 000011000101000110000100011010100100000000000000	10308
19	1101001111000101100010101110010011001000000000000001 000000001100011000000000000000011000000100001000001 1 0010110000111011100000000000010000010000010000000000 100000001010000101000000000101010100001001101000101 0110001000000000000000100000010010011100000000001000 0100000000100010011110010010000100000100010011010000 1000000011000000010000000001000100000000001000000	10308
20	1011100000000111110110111110001110001100001011010111 0110100101100000000100100000011010000001000000000000010 1111110100111000000000100000001000001010001001000000 00010000100010011110011111010000000000000000001000100 0010000000000100000000100001000111110001010100101100 0000001101101101111110000000100000000001000000000000 1000000011000000100000110100001000000000001001000	10308